UNIVERSITÀ DEGLI STUDI DI UDINE
DIPARTIMENTO DI MATEMATICA E INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA
CICLO XXII

PHD THESIS

On the Hybridization of Constraint Programming and Local Search Techniques: Models and Software Tools

Raffaele Cipriano

Supervisors:
Agostino Dovier
Luca Di Gaspero

ANNO ACCADEMICO
2010/2011

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Contents

List of Figures

List of Tables

Introduction

Combinatorial problems such as planning, scheduling, timetabling, and, in general, resource management problems, are encountered daily by industries, corporations, hospitals, and universities. The ability to get quickly good solutions for these problems is essential for such organizations, since it impacts business and overall customer and employee satisfaction. This allows for increasing of profit margins, proper use and employment of materials and resources, optimized utilization of machines and employees, the scheduling of work shifts, and so on.

These problems can be encoded with a mathematical approach that considers multiple elements such as, connections and relationships of entities, prohibited situations, all while working to obtain the main objective that is desired. This mathematical approach is fundamental in the characterization of Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs). A CSP is defined as the problem of associating values, taken from a set of domains, to variables subject to a set of constraints. A solution is an assignment of values to all the variables that satisfies all the constraints. In COPs a cost function is associated to the variable assignments and a solution that (without loss of generality) minimizes the cost value searched. Unfortunately, these problems typically belong to the class of NP-hard (or worse) problems. Thus, in non trivial instances it is extremely unlikely that someone could find an efficient method (i.e., a polynomial algorithm) for solving them exactly.

The number of known approaches for dealing with CSPs and COPs currently is vast, and comparable in size only to the difficulty of this class of problems. These approaches range from mathematical (Operations Research) approaches (e.g., Integer Linear Programming, Column Generation) to AI approaches, such as Constraint Programming (CP), Evolutionary Algorithms, SAT-based techniques, Local Search (LS), just to name a few.

It is also well-known that their intractability poses challenging problems to the programmer. Ad-hoc heuristics that are adequate for one problem are often useless for others [83]. Classical techniques like Integer Linear Programming (ILP) need an extensive tuning on typical problem instances. Any change in the specification of the problem requires restarting almost from scratch. Notwithstanding there is agreement on the importance of developing tools for challenging this kind of problems in an easy and effective way. In this context we are not attempting to find the optimum solution to a problem but instead to find a *high quality solution* to be computed in a reasonable time.

In recent years much effort has been devoted to the development of general techniques that allow high-level primitives to encode search heuristics. Noticeable examples of these techniques are Constraint Programming (CP—with the various labeling heuristics) [61] and Local Search (LS—with the various techniques to choose and exploring the neighborhood). *CP based systems* [2] are usually based on complete methods that analyze the search space by alternating deterministic phases (constraint propagation) and non-deterministic phases (variable assignment). The main advantage of CP is *flexibility*: this paradigm allows the user to encode declarative models in a compact and flexible way, where adding new constraints

is straightforward and does not affect the original model. *LS methods* [1], instead rely on the definition of "proximity" (or neighborhood) and explore only specific areas of the search space. The main advantage of LS methods is *efficiency*: by concentrating on some parts of the search space, they can approximate optimal solutions in shorter time frames.

The future seems to stay in the combinations of these techniques, in order to exploit the best aspect of each technique for the problem at hand (after a tuning of the various parameters on small instances of the problem considered). See for example the results reached in [5, 13, 14, 25, 46, 74]. This is also witnessed in the success of the CP-AI-OR meetings [3]. In particular, Large Neighborhood Search (LNS) can be viewed as a particular heuristic for local search that strongly relies on a constraint solver [19] and that is a reasonable way to blend the inference capabilities of LS and CP techniques.

Aims and Contributions

This thesis aims to study the integration of the Constraint Programming and Local Search paradigms with three main goals in mind. First we want to combine together two free state-of-the-art solvers for LS and CP in order to obtain a single framework for easily developing hybrid techniques thus, exploiting the effectiveness of the basic solvers. Then, we define a meta-heuristic modeling language to model CSPs and COPs as well as to define the meta-heuristic to solve the problem. This language acts as high-level interface to access our framework functionalities. And finally, we test our tool on several benchmark problems, in order to show its effectiveness compared to other solvers for combinatorial problems.

In this thesis we present a general framework called GELATO that integrates CP and LS techniques, defining a general LNS meta-heuristic that can be modified, tuned and adapted to any optimization problem, with a limited programming effort. Instead of building a new solver from scratch, we based our framework on two existing systems: the Gecode CP environment [67] and the LS framework EasyLocal++ [27]. We choose these two systems (among other ones, such as [57, 58, 74]) because both of them are free, open-source, strong, complete, written in C++, easy to interface to/from other systems, and with a growing community using them. Thus, GELATO inherits all the characteristics of these basic tools, providing all their functionalities as well as several new hybrid possibilities. It is worth noting that with GELATO it is possible to define *one single model* to exploit it to perform the search with different strategies. These strategies range from pure Locals Search to pure Constraint Programming through several degrees of hybridization (i.e., LNS strategies), controlled by parameter values. Moreover, GELATO does not require any modification of Gecode and EasyLocal++, not affecting their single development line. In this way every improvement of the basic tools, such as new functionalities or performance improvements, is immediately inherited by GELATO. GELATO is able to read and solve models encoded in Gecode or FlatZinc (a low-level target language for CSPs and COPs).

In order to facilitate the use of the GELATO solver for users not versed in C++, we allow access to the GELATO functionalities from other high-level declarative languages, such as MiniZinc and SICStus Prolog. MiniZinc is a modeling language close to OPL, while SICStus Prolog is the state-of-the-art constraint logic programming language. A model written in MiniZinc or SICStus Prolog is translated into a FlatZinc encoding, that can be read and

executed by the GELATO tool. For the translation task two compilers are used as follows: the minizinc-to-flatzinc compiler, provided by the NICTA research group and a sicstus-to-flatzinc compiler that we developed on our own, starting from the idea developed in [16]. Accessing GELATO from other modeling paradigms is straightforward, since only a compiler from the new paradigm to the FlatZinc language is needed.

Moreover we defined an extension of MiniZinc and SICStus, that allows us to specify the hybrid solving strategy that GELATO will use to solve a model. The user can choose a hybrid schema that performs the search and sets the parameters for guiding the hybrid exploration. However we test the tool on some difficult benchmarks, so as to suggest a methodology to tune the various parameters values on typical problem instances.

Thus, the extended MiniZinc and SICStus languages become *meta-modeling languages* that allow the user to easily model CSPs and COPs and define algorithms that combine cleverly constraint propagation phases, neighborhood exploration and other procedures. Thanks to these meta-modeling languages, GELATO is a *programming framework* made up of three main part: the *modeling part*, where the user will define in a high-level style the problem and the instance he wants to solve as well as the algorithm to use. Algorithms used may be CP search, possibly interleaved with LS, heuristics or meta-heuristics phases. The *translation part*, where the model and the meta-algorithm defined by the user will be automatically compiled into the a FlatZinc format. Finally, the *solving part*, where the FlatZinc model will be run and the various solvers will interact as specified by the user in the first phase, in order to find the solution for the instance of the problem modeled. As a side effect, this programming framework represents a *new way to run declarative models*: the user can encode problems using well-known declarative languages (e.g. Prolog, MiniZinc), and then make use of new low-level implementations (e.g. Gecode solver) for efficient execution. It is worth noting that we also allow using GELATO from models directly encoded in Gecode or FlatZinc. Thus, GELATO can be used by any language that have a back end to the Gecode environment, as the one provided for the Haskell language in [84].

In order to tune our framework and show its effectiveness, we tested GELATO on the following three benchmarks:

- the Asymmetric Travel Salesman Problem, the asymmetric version of the well known TSP NP-hard problem, taken from the TSPLib [72];
- the Minimum Energy Broadcast problem, an NP-hard optimization problem for the design of ad hoc Wireless Networks, drawn from CSPLib [40] (number 48);
- the Course Time Tabling problem, consisting of the weekly scheduling of the lectures for several university courses subjected to several given restrictions. This problem has been introduced as Track 3 of the second International Timetabling Competition held in 2007 [49].

We compared the performance of the GELATO hybrid approach with the approaches of pure Constraint Programming and pure Local Search as well as with the state-of-the-art framework for hybrid methods, i.e., the Comet language. The results show that GELATO has better performance with respect to the basic tools as well as the Comet language. With respect to Comet, GELATO is free, open source, and easy to interface, being entirely

made of C++ classes. Moreover, a user can model any problem choosing from high-level declarative languages (e.g. SICStus Prolog, MiniZinc, Haskell) and exploit the available compilers to obtain Gecode of FlatZinc encoding that can be directly used by GELATO.

Thesis Organization

The thesis is divided into three main parts. The first part gives an overview of the basic concepts of modeling combinatorial optimization problems and introduces the Constraint Programming and the Local Search paradigms, as well as the approaches that hybridize these two techniques. The second part presents the software tools, i.e., Gecode and EasyLocal++ as well as the framework GELATO, together with its meta-heuristics modeling language. The third one explains the experiments performed on the three benchmarks, showing and comparing the results obtained.

Specifically, Chapter 1 explains the basic concepts of combinatorial optimization, i.e., Constraint Satisfaction Problems and Constraint Optimization Problems, introducing terminology and examples used throughout the thesis. The Constraint Programming approach to solve combinatorial problems is described in Chapter 2. Then, Chapter 3 introduces the family of Local Search algorithms and meta-heuristics. The first part is finalized giving an overview of the possible hybridization among the CP and LS paradigms in Chapter 4. This chapter especially focuses on Large Neighborhood Search (LNS), a particular LS approach, that naturally leads to the definition of hybrid algorithms, and thus heavily used in GELATO and Comet. Chapter 4 also includes a brief overview of the Comet programming language.

The second part of this thesis deals with the software tools used and developed to tackle combinatorial problems. The Gecode programming environment, i.e., our basic Constraint Programming solver is introduced in Chapter 5. The EasyLocal++ framework, used for managing Local Search algorithms, is shown in Chapter 6. Then Chapter 7 deals with GELATO, the framework we developed combining together the functionalities of Gecode and EasyLocal++. The overall architecture and the components of GELATO are explained as well as how the components interacts together and with the basic solvers. Brief examples of how to use the framework are also given. Chapter 8 presents information on how to access the GELATO functionalities from an high-level declarative language. The SICStus Prolog and the FlatZinc languages are introduced as well as the compiler we developed to translate SICStus Prolog models into FlatZinc ones. Moreover, in this chapter the extensions of SICStus Prolog and FlatZinc to define meta-heuristics are presented. Finally, Chapter 9 is a technical chapter explaining the steps needed to install and use the basic solvers and GELATO into a machine.

The third part of the thesis focuses on the experiments. The experimental methodology is introduced in Chapter 10, that also deals with tests performed on the ATSP benchmark. The experiments performed on the MEB problem are reported in Chapter 11, and Chapter 12 focuses on the CTT tests.

In conclusion of this thesis we give some commentary about this research and describe possible future research to further develop that starting in this work.

I

Preliminary Concepts

1

Constraint Satisfaction and Optimization Problems

In this chapter we give the basic definitions and characterizations of Constraint Satisfaction Problems (CSPs) and Constraint Optimization Problems (COPs). CSPs and COPs allow to mathematically model real life problems and they can be solved with a lot of different techniques.

These problems are specified by a set of *variables* and a set of associated *domains* that together define the *search space*, i.e. the set of all the possible variable assignments according to the domains. A set of *constraints* distinguish between solutions (i.e., assignments that satisfy the constraints) and assignments (that violate some constraints). The constraints guide the search process discarding parts of the search space that violates constraints, thus driving the search to solutions satisfying the constraints. The above characteristics are common between CSPs and COPs. A COP has the additional definition of an *objective function* that measures the goodness of a single solution and drives the search process discovering good solutions (instead o just admissible ones).

1.1 Formal definitions for CSP and COP

Formally, a CSP $\mathcal{P}$ is defined by triple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, C \rangle$, where:

- $\mathcal{X}$ represent a finite set of *variables* $\mathcal{X} = \{x_1, \ldots, x_k\}$. They usually model the objects of the problem, e.g., resources, machines, personnel, time-slots, space-coordinates and so on.

- $\mathcal{D}$ is a set of *domains*. Each variable $x_i \in \mathcal{X}$ has associated a domain D_i that models all the possible values that the variable can take (i.e., if $x_i = d_i$ then $d_i \in D_i$). The set of all the domains is $\mathcal{D} = \{D_1, \ldots, D_k\}$. The domains are usually subset of $\mathbb{R}$, $\mathbb{N}$, $\mathbb{Z}$ or the set $\{true, false\}$.

- C is the set of all the *constraints*. A constraint is a relation over $\mathbf{dom} = D_1 \times \cdots \times D_k$. The set of all the constraints is $C = \{C_1, \ldots, C_n\}$. A tuple $\langle d_1, \ldots, d_k \rangle \in \mathbf{dom}$ *satisfies*

a constraint $C_i \in C$ iff $\langle d_1, \ldots, d_k \rangle \in C_i$; a tuple $\langle d_1, \ldots, d_k \rangle \in \mathbf{dom}$ satisfies C if it satisfies all $C_i \in C$.

The *search space* of the CSP $\mathcal{P}$ represents the set of all the possible assignments $d = \langle d_1, \ldots, d_k \rangle \in \mathbf{dom}$ for the variables $\langle x_1, \ldots, x_k \rangle$ of the CSP $\mathcal{P}$. d is a *solution* of a CSP $\mathcal{P}$ if $\forall i$, $x_i = d_i$, $d_i \in D_i$ and if moreover d satisfies every constraint $C \in C$. In some contexts, a solution is referred with the name of feasible solution, to stress the *feasibility* concept, i.e. that all the constraints are satisfied. If there exists a constraint $C_j \in C$ so that d doesn't satisfy C_j, then d is not a solution but just an assignment. With abuse of notation, assignments violating constraints are sometimes called unfeasible solutions. The set of solutions of a CSP $\mathcal{P}$ is denoted by $\mathbf{sol}(\mathcal{P})$. If $\mathbf{sol}(\mathcal{P}) \neq \emptyset$, then $\mathcal{P}$ is *consistent*.

A COP O is defined by two elements, i.e., $O = \langle \mathcal{P}, f \rangle$, where:

- $\mathcal{P}$ is a CSP as defined in the previous paragraphs;
- f is an *objective functions* associated to $\mathcal{P}$. The objective function is defined as $f : \mathbf{sol}(\mathcal{P}) \to E$ where $\langle E, \leq \rangle$ is a well-ordered set (typically, $E \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$).

The search space of O is the set $\mathbf{sol}(\mathcal{P})$ and it is usually assumed to be non-empty for COPs. When clear from the context, we write $\mathbf{sol}(O)$ for $\mathbf{sol}(\mathcal{P})$. $d \in \mathbf{sol}(\mathcal{P})$ is a *optimal solution* for O if it *minimizes* the function f, namely $\forall e \in \mathbf{sol}(\mathcal{P})(f(d) \leq f(e))$. We remark that even if we are modeling a problem that has the conceptual aim to *maximize* a certain function, we can always model it using a *minimizing objective function*: so, from now on, when talking about COPs, we always refer to minimization problems.

The objective function can also be used to define CSPs: the constraints are relaxed and f is used as a *distance to feasibility*, which accounts for the number of relaxed constraints that are violated. The constraints that may be violated and contribute to worse the objective function value, are called *soft constraints*; on the contrary, the constraints that cannot be violated in any case are called *hard constraints*.

CSPs and COPs can be solved with several techniques, coming from different areas of the AI and Operation Research fields, such as Simplex Method [21, 22], Branch and Bound [47], Branch and Cut [38, 59], column generation [20, 23, 34], Boolean Satisfiability [17, 39], Constraint Programming [2], heuristics of Local Search [1], Genetics Algorithms [41] and so on. In the rest of the thesis we focus on Constraint Programming and Local Search methods.

1.2 Example

In this section some examples that clarify the above definitions are given. Let us consider a simplified version of the Course Time Tabling problem, that will be presented in its complete version in chapter 12. It consists in the weekly scheduling of the lectures of a set of university courses and here it is called SCTT (i.e., Simple Course Time Tabling problem) and is defined as follows.

DEFINITION 1.2.1 (SCTT) *Given a set of courses $C = \{c_1, \ldots, c_n\}$, each course c_i is given in a number of weekly lectures $l : C \to \mathbb{N}$, and is taught by a teacher $t : C \to T = \{t_1, \ldots, t_g\}$. Five teaching days (from Monday to Friday) and two time slots for each day (Morning and*

Afternoon) are given, thus having a set of ten possible time periods: $P = \{p_1, \ldots, p_{10}\}$. *Each teacher* t_i *can be unavailable for some periods* $u : T \to 2^P$.

The problem consists in finding a schedule for all the courses, so that all the required lectures for each course are given (C1), lectures of courses taught by the same teacher are scheduled in distinct periods (C2) and teacher unavailabilities are taken into account (C3).

Moreover, a cost function is defined for the following criterion: the lectures of each course should be at least spread into a given minimum number of days $\delta : C \to \{1, \ldots, 5\}$.

Let us consider the following instance:

- $C = \{OS, PL, AI\}$
- $l(OS) = 3$, $l(PL) = 4$, $l(AI) = 3$;
- $t(OS) = Turing$, $t(PL) = Turing$, $t(AI) = Asimov$
- $P = \{MO_m, MO_a, TU_m, TU_a, WE_m, WE_a, TH_m, TH_a, FR_m, FR_a\}$
- $\delta(OS) = 3$, $\delta(PL) = 4$, $\delta(AI) = 3$,
- $u(Turing) = \{TU_m, TU_a\}$, $u(Asimov) = \{TH_m, TH_a, FR_m, FR_a\}$

A possible model for this problem can be defined in this way:

Variables and Domains The set $\mathcal{X}$ is made by $C \cdot P$ variables $x_{c,p}$ having domains $\{0, 1\}$. The intuitive meaning is that $x_{c,p} = 1$ if and only if course c is scheduled at period p, and $x_{c,p} = 0$ if course c is not scheduled at period p.

Constraints To ensure that the required number of lectures is given for each course, the following constraint is introduced:

$$\sum_{j \in P} x_{c,j} = l(c) \quad \forall c \in C \tag{C1}$$

The constraints stating that lectures of courses taught by the same teacher are scheduled in distinct periods can be modeled as:

$$x_{c,j} \cdot x_{c',j} = 0 \quad \forall c, c' \in C \text{ s.t. } t(c) = t(c'), j \in P \tag{C2}$$

The unavailabilities constraints are modeled as follows:

$$x_{c,j} = 0 \quad \forall j \in u(t(c)), c \in C \tag{C3}$$

Periods →	MO		TU		WE		TH		FR		$fObj$
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	1	0	0	0	1	0	1	0	0	0	3 − 3 = 0
PL	0	0	0	0	0	1	0	1	1	1	4 − 3 = 1
AI	1	1	1	0	0	0	0	0	0	0	3 − 2 = 1
							Total $fObj$:				2

Figure 1.1: Solution for the given instance of SCTT

Periods →	MO		TU		WE		TH		FR		$fObj$
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	1	0	0	0	1	0	1	0	0	0	3-3=0
PL	1	0	0	0	0	1	0	1	0	1	4-4=0
AI	0	0	0	0	1	1	1	0	0	0	3-2=1
							Total $fObj$:				1

Figure 1.2: Assignment for the given instance of SCTT, violating constraints C2 and C3

Search space and solutions In this instance we have 30 variables (10 time slots × 3 courses) with Boolean domains. Thus, an assignment can be represented by a sequence of 30 bits. The entire search space is made of all the possible sequences, thus containing 2^{30} possible assignments. Not all these assignments are solutions, because of the constraints. Figures 1.1 and 1.2 show respectively a solution and an assignment. The assignment violates the constraints C2 and C3: on Monday morning Prof. Turing has two different lectures in the same time slot and Prof. Asimov gives a lecture on Thursday, when he is unavailable. The solution and the assignment can be respectively represented by the following sequences: 100010100000000101111110000000 and 100010100010000101010000111000.

Objective function The objective function is defined as follows: each course is supposed to be spread into a minimum number of days, according to the δ function. So, for each course c the difference between $\delta(c)$ and the actual number of days occupied by the lectures of c is calculated. This gives the contribution of each course to the objective function. The overall value of the objective function is then calculated as the sum of all the single contributions. The formula for such objective function is:

$$fObj = \sum_{i=1}^{n} \max(0, \delta(c_i) - |\{d(j) \ : \ x_{c_i,j} > 0\}|) \tag{1.2.1}$$

According to this objective function, the solution of figure 1.1 is not optimal. An optimal solution is shown in figure 1.3.

Periods →	MO		TU		WE		TH		FR		$fObj$
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	1	0	0	0	1	0	1	0	0	0	3-3=0
PL	0	1	0	0	0	1	0	1	0	1	4-4=0
AI	1	0	1	0	1	0	0	0	0	0	3-3=0
							Total $fObj$:				0

Figure 1.3: Optimum solution for the given instance of SCTT

2

Constraint Programming

Constraint Programming (CP) [61] is a declarative programming methodology, parametric on the constraint domain. The constraint domain can be of several kinds, such as real, finite, interval and Boolean domains. From now on in this thesis, we only consider Constraint Programming on *finite domains*, because combinatorial problems are usually encoded using constraints over finite domains, currently supported by all CP systems (e.g., [57, 58, 67]). The CP paradigm is usually based on *complete methods*, i.e., methods that systematically analyze the whole search space, looking for a solution. The search process acts alternating two phases:

1. a deterministic phase, called *constraint propagation*, where the values that cannot be assigned to any solution are removed by domains;

2. a non-deterministic phase, called *variable assignment*, where a variable is selected and a value from its domain is assigned to it.

These two phases are iterated until a solution is found or unsatisfiability is reached; in case of unsatisfiability the process backtracks to the last choice point (i.e., the last variable assignment) and tries other assignments. The control algorithms that execute the searching process alternating these two phases is called *search engine*.

In the following sections we explain in more details these two phases and the most common search engines.

2.1 Constraint Propagation

Given a CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, C \rangle$, the constraint propagation process rewrites a constraint $C \in C$ or a domain $D \in \mathcal{D}$ into an equivalent one applying rewriting rules that satisfy some local consistency properties. There are three types of rewriting rules: *Domain Reduction Rules*, that reduce the domains exploiting the information carried by constraints; *Transformation Rules*, that transform the constraints into a simple form; *Introduction Rules*, that add new constraints implied by the existing ones. Constraint programming solvers usually implement fast Domain Reduction Rules and the most common ones are *node consistency*, *arc consistency*, and *bounds consistency*.

Node Consistency A CSP is node consistent if for each variable $x_i \in \mathcal{X}$, every unary constraint that involves x_i is satisfied by all values in the domain D_i of the variable x_i and vice versa. This condition can be trivially enforced by reducing the domain of each variable to the values that satisfy all unary constraints on that variable. As a result, unary constraints can be neglected and assumed incorporated into the domains.

Arc Consistency A binary constraint C on the variables x_i and x_j with domains D_i and D_j is arc consistent if each admissible values of x_i is consistent with some admissible value of the x_j and vice versa. Formally:

$$(\forall a \in D_i)(\exists b \in D_j)((a,b) \in C) \wedge (\forall b \in D_j)(\exists a \in D_i)((a,b) \in C).$$

A CSP is arc consistent if every binary constraint C is arc consistent. The arc consistency property may be very expensive to implement, so many solvers approximate it guaranteeing just the following notion of consistency.

Bounds Consistency It is an approximation of the arc consistency, since it checks only the bounds of the minimal interval that includes the domain. However, this is correct only if the type of the domain is an interval. In the general case (e.g., large domains with few elements and holes in the intervals) the bounds consistency is not complete w.r.t. arc inconsistency. Even if it is an approximation it is deeply used in the Constraint Programming system, because it is very efficient.

We define the two functions *min* and *max* that return, respectively, the minimum and the maximum element of a domain D. A binary constraint C over variables x_i and x_j with domains D_i and D_j is bounds consistent if:

1. $(\exists b \in \{min(D_j)..max(D_j)\})((min(D_i), b) \in C) \wedge$
   $(\exists b \in \{min(D_j)..max(D_j)\})((max(D_i), b) \in C)$, and
2. $(\exists a \in \{min(D_i)..max(D_i)\})((a, min(D_j) \in C) \wedge$
   $(\exists a \in \{min(D_i)..max(D_i)\})((a, max(D_j) \in C)$

A CSP is bounds consistent if every binary constraint in C is bounds consistent.

Now we give a brief example showing the different computational complexity needed to guarantee arc consistency and bounds consistency. Consider c binary constraints $C_1, \ldots, C_c$ on couples of variables from $x_1, \ldots, x_k$ with domains in $D_1, \ldots, D_k$. Let d be the maximum cardinality among the domains. Let us analyze the *worst case* scenario, for both arc consistency and bounds consistency checking algorithms. Checking the arc consistency of a generic constraint $C_k(x_i, x_j)$ means to check that for every variable in the constraint (2 variables, i.e., x_i and x_j), for every domain value (at most d) in D_i there exists an admissible value for x_j in D_j (again D_j has at most d elements). Thus, having c constraints, at each step the algorithm has to check a number of values equal to $c \cdot 2 \cdot d \cdot d = 2cd^2$, i.e., $O(cd^2)$. The number of overall steps is bounded by kd (at each step at least one values from one domain is discarded), leading to an overall complexity of $O(kcd^3)$. An algorithm that guarantees bounds consistency does not check all the values of D_i, but just the lower and the upper bounds, for a number of values equal at most to $4d$. Having c constraints the values checked at each step are at most

c, so at most $4dc$ checks. The number of overall steps is bounded by kd, leading to an overall complexity of $O(kcd)$.

Generalization of Arc and Bounds Consistency Arc consistency as well as bounds consistency can be generalized to non binary constraints, leading to the notion of *hyper arc consistency* and *hyper bounds consistency*.

An n-ary constraint C on the variables $x_1 \dots x_n$ is *hyper arc consistent* if $\forall i = 1, \dots, n$, the following property holds:

$$(\forall a_i \in D_i)(\exists a_1 \in D_1)\dots(\exists a_{i-1} \in D_{i-1})(\exists a_{i+1} \in D_{i+1})\dots(\exists a_n \in D_n)|(\langle a_1 \dots a_n \rangle \in C).$$

A CSP is *hyper arc consistent* if all the constraints in C are hyper arc consistent.

In the same way we can generalize the notion of bounds consistency to *hyper bounds consistency*. Moreover, other families of local consistency can be defined (see [2]). For instance, the *path* consistency, that involves triplets of variables; the *k-consistency*, that generalize the previous notions of consistency in a recursive fashion; the *directional consistency*, a variant of arc, path, and k-consistency tailored for being used by an algorithm that assigns values to variables following a given order of variables.

Checking the consistency of $\mathcal{P}$ Once the desired level of consistency is chosen, a propagation algorithm must ensure that consistency is preserved, i.e., it checks if the consistency properties holds and discards the domain values that are inconsistent. These algorithms are called *propagators* and usually every constraint has associated its own propagator. In the deterministic propagation phases of the constraint programming search, all the propagators are activated one by one, and iterated until the domains cannot be restricted anymore (a *fix-point* has been reached).

2.2 Variable Assignment

Once propagation reaches the fix-point and cannot infer any more information, it is necessary to start a non-deterministic phase that explores the search space of $\mathcal{P}$. This non-deterministic phase, also called *branching*, first applies a *splitting rule*, that splits the search space in two or more sub-spaces, and then proceeds exploring one of the sub-spaces. We remark that we are dealing with CSPs over finite domains, thus it is always possible to perform an enumeration of the domains. The most common splitting rules concern *domain splitting*:

1. domain labeling:
$$\frac{x \in \{d_1, \dots d_n\}}{x \in \{d_1\}|\cdots|x \in \{d_n\}}$$
2. domain enumeration
$$\frac{x \in \mathcal{D}}{x \in \{d\}|x \in \mathcal{D} \setminus \{d\}}$$
with $d \in \mathcal{D}$

3. domain bisection

$$\frac{x \in \mathcal{D}}{x \in \{min(\mathcal{D})..a\} | x \in \{b..max(\mathcal{D})\}}$$

where $a, b \in \mathcal{D}$ and b is the greater element after a in $\mathcal{D}$. If $\mathcal{D}$ is an interval $x \ldots y$, we can choose $a = \lfloor (x + y)/2 \rfloor$ and $b = a + 1$.

If a sub-space is discovered to have no solutions, the process *backtracks* to the previous choice point (splitting rule) and chooses another branch (sub-space) of the splitting. If all the sub-spaces have been already explored without success, it backtracks further to the previous splitting point and choose another branch, and so on.

The most common rule (implemented in all the constraint programming solvers) is the *domain labeling*. To fully define the splitting process based on this rule, we have to specify two aspects: the *variable selection* and the *value selection*. The *variable selection* rule defines the next (still unassigned) variable x_i that will be chosen for the domain splitting rule. The *value selection* rule defines the criterion to choose in which order the different possible domain values of D_i will be assigned to x_i (i.e., in which order the sub-space generated by the splitting rule will be explored). Different selection strategies (concerning variables and values selection) can drastically effect the search efficiency of a constraint programming algorithm and its results on a specific problem instance.

2.3 Search Engines

A search engine is the control algorithm that actually performs the search. It alternates the constraint propagation and space splitting phases explained in the previous sections. The engine explores the search space, until a solution is found or the whole search space is explored without finding any solution, thus showing that the problem is inconsistent. There are several kinds of search engines, each with different characteristics to assess different problems. The next subsections present the search engines commonly available in the constraint programming environments.

2.3.1 Depth first

The depth first engine is used for CSPs and explores the search tree of a problem with a depth-first algorithm. It starts on the root node of the search tree, where all the variables are not instantiated and all the domains are complete. The algorithm first runs a propagation on the root node. Then it starts the proper exploration, alternating domain labeling and constraint propagation phases. At each step, according to the variable and the value selection strategy chosen, a new subproblem is created. Then a constraint propagation phase is applied to this new subproblem. After this propagation phase, if the subproblem is not yet completely solved (i.e., all the variables are instantiated), or it is discovered to be inconsistent, the exploration continues in the subproblem, with a new domain labeling step. When the current subproblem is completely solved, the algorithm stops and the solution found is returned. If instead the current subproblem is discovered to be inconsistent, the algorithm backtracks to the previous domain labeling phases and assigns another value to the variable, according to the value

selection policy, thus creating a new subproblem to be explored. If all the possible values have been tried without success, the algorithms backtracks to a previous domain labeling choice points and tries another value for that variable.

2.3.2 Branch and bound

This engine is used on COPs, in order to find optimal solutions. It performs the exploration similarly to the depth first search engine, but it does not stop at the first solution (as the depth first search engine); instead it keeps searching for better solutions, until a stop criterion is satisfied. When this engine finds a solution, it adds a new constraint to the problem, in order to make the solution found unfeasible and drive the search towards better solutions.

The most common example of such a constraint is as follows. The COP $\mathcal{P}$ is a minimization problem, with the objective function defined on the variable $FOBJ$. The solution s has been found, with objective function value $f(s) = v$, so the constrains store contains the information $FOBJ = v$. The constraint that discards the current solution and force the search process to better one is $FOBJ \leq v$. This constraint not only forces the search towards new solutions, but also cuts portions of the search space. In fact, if it turns out that part of the search tree does not have any chance to improve the objective function value already reached, this subspace is skipped. The general idea is the same of the Operation Research Branch and Bound method: an analysis of the lower bound for a partial assignment, together with a know value already reached, allows the algorithm to cut parts of the search tree and speed up the search. However, the implementation of the OR Branch and Bound method is different from the CP one, since it uses concepts typical of the OR field, such as linear relaxations, the simplex method, particular branching strategies and bounding methods.

Thus, at each solution found the branch and bound search engine adds a new constraint and starts backtracking to find another (better) solution. The backtrack process is performed in a depth first fashion, as described for the previous search engine.

The branch and bound search engine cannot detect if a solution found is the optimal one, as long as a lower bound for the current instance is not known. Thus, the engine needs a stop criterion, in order to stop the search process at some point and return the best solution found so far. There are different kinds of stop criterions, depending on the constraint programming environment we are using. The most common ones are: timeout, failures, subproblems, memory. The *timeout* criterion stops the search engines after a given amount of time. The *failures* criterion stops the search when a given number of subproblems has been discarded because of inconsistency. The *subproblems* one stops the search after the exploration of a given number of subproblems. The *memory* one stops the search when a given amount of memory has been used.

2.3.3 Depth first with restart

This search engine is used on COPs and it is very similar to the branch and bound one. The engine explores the search space in a depth-first way, then when it finds a solution it adds a new constraint that discards the solution and forces the engine to search for a better ones. The engine subsequently stops when a specified stop criterion is satisfied. The only difference compared to the previous search engine is that, after it has found a solution and added the

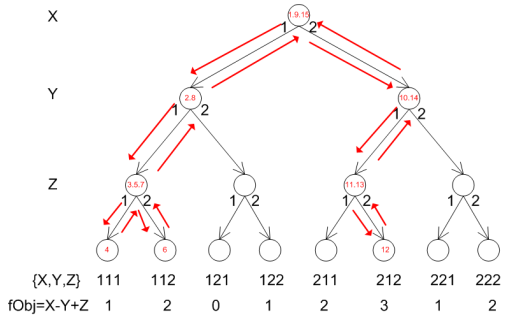


Figure 2.1: Example of Branch and Bound search

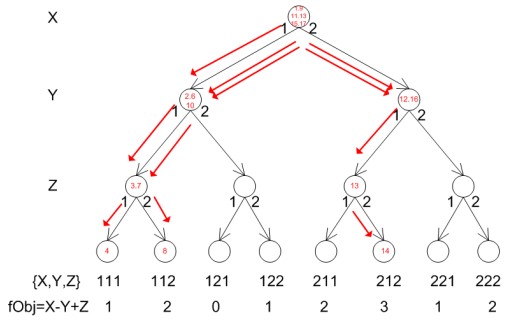


Figure 2.2: Example of Depth First with Restart search

new constraint, it does not start to backtrack. Instead, it starts the search from the root node of the search tree, i.e. from the beginning. The new constraint added in the constraints store should now cut the search space and drive the search to different (and better) solution.

Now a brief example that shows the different explorations of a search space performed by a branch and bound as well as a depth first with restart search engine is given. Consider the following COP: $\mathcal{X} = \{X, Y, Z\}$, $D_X = D_Y = D_Z = \{1, 2\}$, $C = \emptyset$, $fObj = X - Y + Z$. In this example the objective is to maximize $fObj$. Note that there are no constraint in the original CSP, but the constraint related to the objective function will be added every time a solution is found, according to the search strategy described above.

Figure 2.1 and 2.2 show the explorations performed by the two different engines. Arrows shows how the search space is explored and the numbers into the nodes represent the visiting order. Note that every time a solution is found (let us say with value f), a constraint $X-Y+Z \leq f$ is added. This constraint is checked for consistency at every node subsequently analyzed, both when backtracking and when restarting. In this example the branch and bound engine shows a better behavior: it finds more quickly the second solution and finishes the exploration in less steps. Anyway, a crucial aspect that we cannot consider here is about how much time is spent on a single node. Restart engines usually come with procedures able to recreate or clone already visited nodes in very short time, making the restart strategy worthy. There is

not a general rule stating when a strategy is better than the other one. It strongly depends on the problem instance we are solving, on the variable ordering, on the time spent to create a node or check consistency, on the strength of the consistency level chosen, and so on.

2.4 Example

In this section explorations of two searches performed by a depth first engine are given. The instance of the SCTT problem introduced in section 1.2 is considered. The states reached by the explorations are shown in figures 2.3 and 2.4. The structure of a state is the same one used in figures 1.1-1.3. Only the grid of the variables are reported, without any labels nor the FObj column.

The explorations are executed by a depth first engine that:

- uses a domain labeling rule for the variable assignment;
- checks and propagates unary and binary constraints every time one of the variables involved in the constraint is instantiated;
- checks and propagates n-ary constraints only when all the variables involved in the constraint are instantiated;

This example simplifies the real behavior of the modern CP engines, that usually propagate and check n-ary constraints also on partial assignments.

The two explorations differ only in the domain labeling strategy used: the first exploration, shown in figure 2.3, uses a *leftmost* variable selection with a *down-up* value selection. The second exploration, shown in figure 2.4, uses a with *leftmost* variable selection with an *up-down* value selection. This difference leads to an ineffective (*down-up* value selection) and effective (*up-down* value selection) exploration of the search tree. Let's briefly analyze these two explorations.

The first exploration (figure 2.3) starts on the root node (1) which has all the variable not instantiated. The first propagation phase fixes to 0 the variables regarding the unavailability constraints, as shown in node (2). Then the first variable is assigned with the value 0, according to the domain labeling strategy (3). After this assignment there is an ineffective propagation, because the assignment of 0 dose not give any helpful information that may restrict some other domains. Then all the variables of the first row are instantiated in the same way, reaching the configuration of node (5) after multiple steps. At this point all the variables regarding the constraint $C1$ for the first course ($C1$ with $c = OS$) are instantiated (from now on we write $C1_{OS}$ to indicate the $C1$ constraint on the OS course). The consistency checking is performed and it detects that constraint $C1_{OS}$ is violated. So the process backtracks to the previous choice point, i.e., node (4) and chooses the other branch. This alternative branch still leads to an inconsistent state (node 6), w.r.t. the constraint $C1_{OS}$. So the engine continues backtracking, until it reaches node (7), which satisfies the constraint $C1_{OS}$. Note that the assignment performed just before node (7), instantiates the last variables of the first row. Thus, the subsequent propagation phases fixes the value 0 on the above variable, according do the binary constraint $C2$ on those variables. Similarly, the other two variables of the second row

were fixed to 0. After node (9), the engine will continue the search and will encounter similar problems (inconsistency forcing backtrack) on the second and the third row.

The second exploration (figure 2.4) shares nodes 1 and 2 with the first one, but then it differs. At the variable assignment step, the first variables gets the value 1, as shown in node (3). At this point the information of the first variable is propagated and the above variable on row 2 gets the value 0. The process continues in this way (assigning 1 to the next variable and propagating 0 above), till node (7) is reached. At this point constraint $C1_{OS}$ is satisfied and the propagation process fixes the other variables to 0. The engines continues the search, reaching the feasible solution represented in node (9) without any backtrack.

The second exploration reaches a solution in fewer steps than the first one, so the up-down value selection is more effective. This occurs because this strategy makes choices that: 1) propagate more information 2) satisfy earlier the constraints.

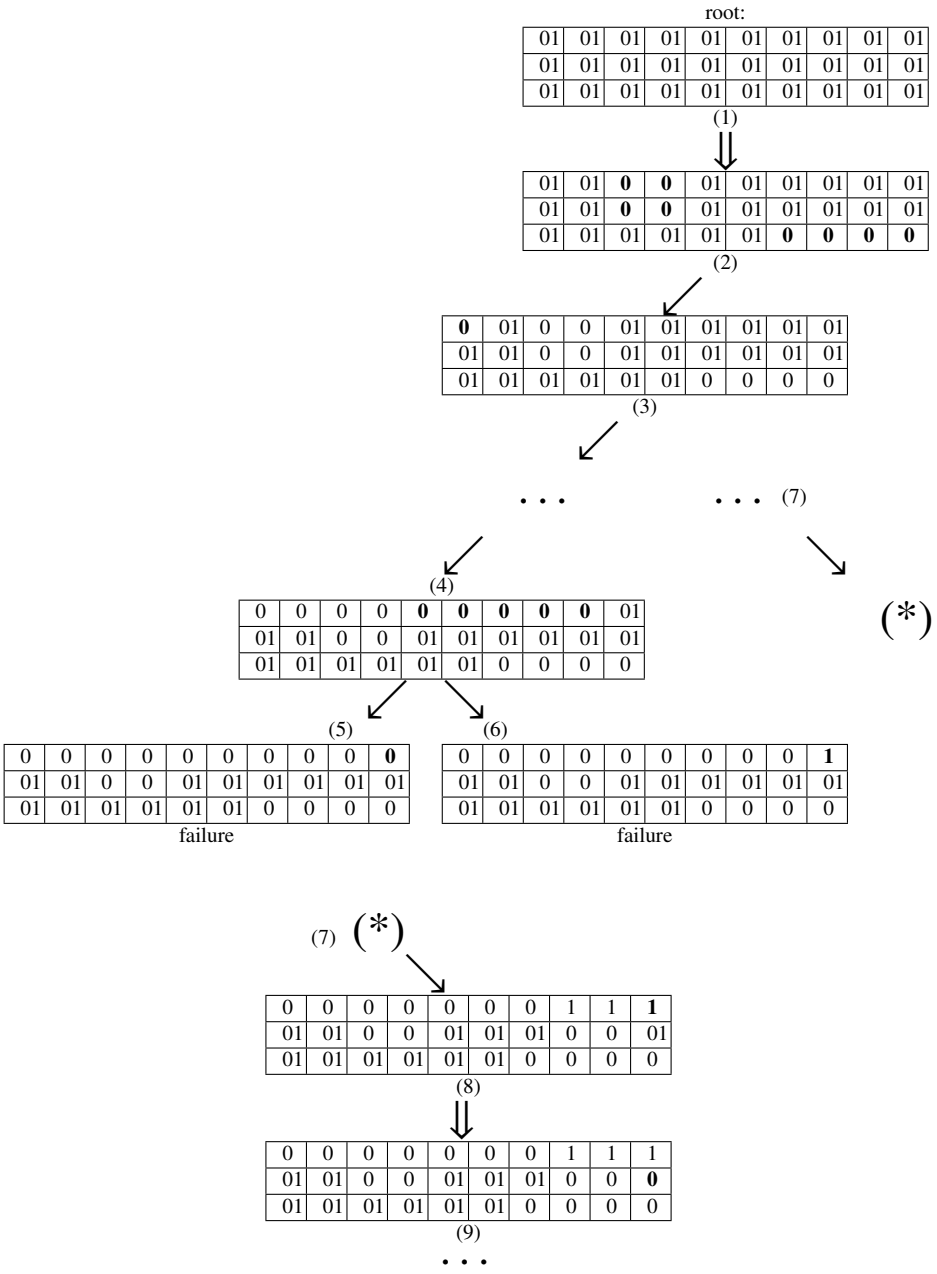


Figure 2.3: Ineffective exploration of the search space for the SCTT problem

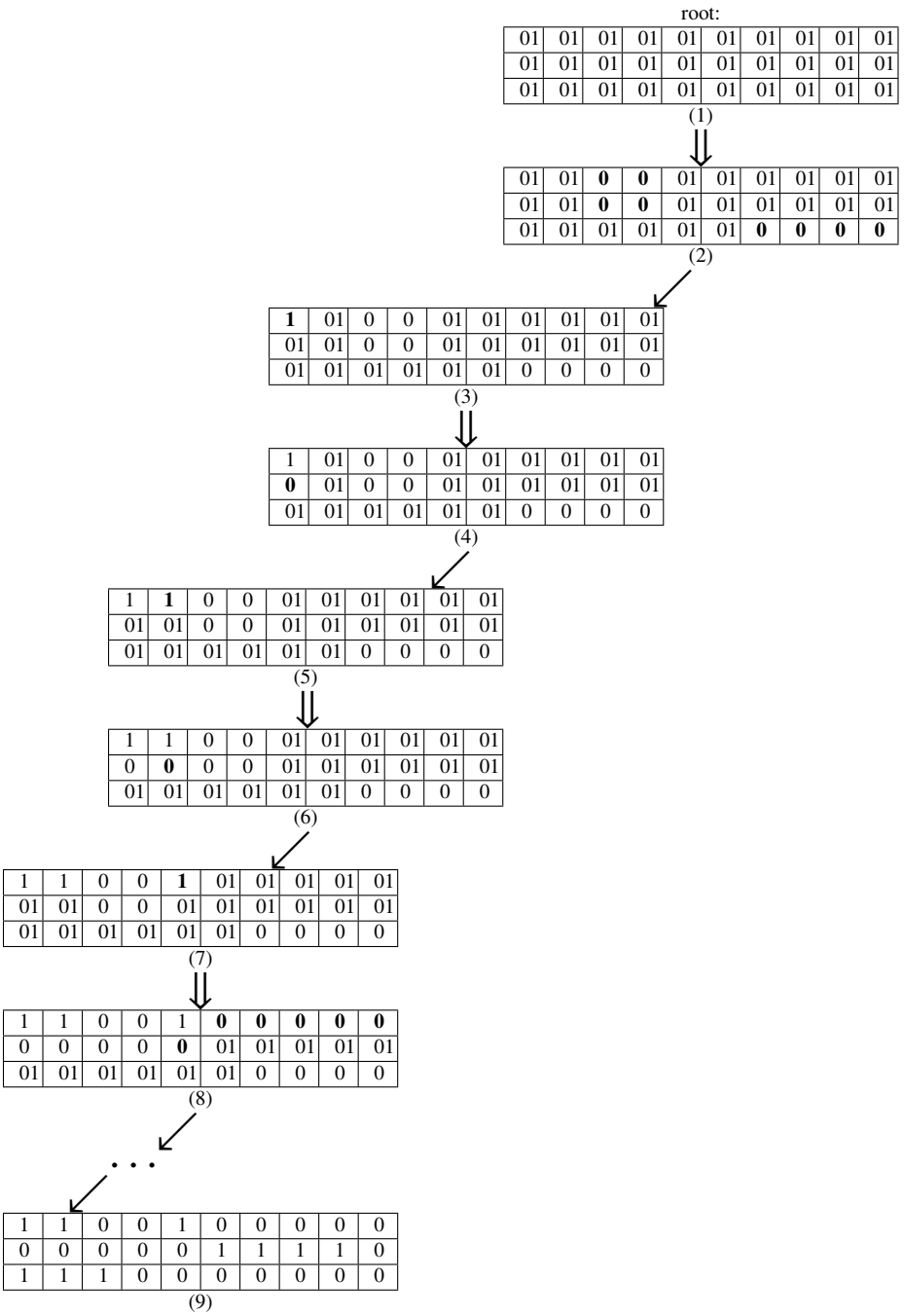


Figure 2.4: Effective exploration of the search space for the SCTT problem

3

Local Search

Local Search (LS) methods (see [1, 24, 42]) are a family of meta-heuristics to solve CSPs and COPs, based on the definition of *proximity* (or *neighborhood*): a LS algorithm typically moves from a solution to a near one trying to improve the objective function, in an iterative process. Differently from the CP paradigm, LS algorithms usually do not explore the whole search space, but they focus only on specific areas of the search space: for this reason they are *incomplete methods*, in the sense that they do not guarantee to find a solution, but they search non-systematically until a specific stop criterion is satisfied. However, from the theoretical point of view there are also local search techniques that are asymptotically complete. They are called *ergodic* techniques [6, 7, 79], which means that at infinite time they will analyze all the possible solutions of a problem, thus exploring the whole search space.

Despite their incompleteness, these techniques are very appealing because of their effectiveness and their widespread applicability.

To specify a *LS algorithm* four aspects must be defined:

- a *search space*, that defines the solutions of the problems;
- a *neighborhood relation*, that defines how to move from a solution to another one, usually called a neighbor solution, or simply a neighbor;
- a *cost function*, that assesses the quality of a solution;
- a *stop criterion*, that establish when the local search algorithm has to stop the search and return the best solution found that far.

In the next sections we illustrate these entities (section 3.1) and show how they can be composed to define different basic LS algorithms (section 3.2). In section 3.3 we explain different ways to improve the basic algorithms. Section 3.4 presents other meta-heuristics that deal with combinatorial optimization problems, belonging to the population-based approaches. The last section (3.5) gives some examples about the local search concepts introduced in section 3.1.

3.1 Local Search Entities

Given a COP $O = \langle \mathcal{P}, f \rangle$, where $\mathcal{P}$ is the CSP $\langle \mathcal{X}, \mathcal{D}, C \rangle$, the *search space* for a local search algorithm is the set $\mathbf{sol}(O)$, as defined in section 1.1. The constraints in $\mathcal{D}$ are usually called hard constraints, because they cannot be violated in any case. Each element $s \in \mathbf{sol}(O)$ represents a solution of P. Local Search methods are based on the hypothesis that $\mathbf{sol}(O)$ is not empty.

Definition 1 *(Neighborhood relation) Given a COP $O = \langle \mathcal{P}, f \rangle$, we assign to each element $s \in \mathbf{sol}(O)$, a set $\mathcal{N}(s) \subseteq \mathbf{sol}(O)$ of neighboring solutions of s. The set $\mathcal{N}(s)$ is called the* neighborhood *of s and each member $s' \in \mathcal{N}(s)$ is called a* neighbor *of s.*

In general $\mathcal{N}(s)$ is implicitly defined by referring to a set of possible *moves*, which define transitions between solutions. Moves are usually defined in an intensional fashion, as local modifications of some part of s.

As defined in section 1.1, a *cost function* $f : \mathbf{sol}(O) \to E$ (where $\langle E, \leq \rangle$ is a well-ordered set) associates each element $s \in \mathbf{sol}(O)$ to a value $f(s) \in E$ that assesses the quality of the solution. The function f is used to drive the search toward good solutions and to select the move to perform at each step of the search. Moreover, f counts the number of violations of the soft-constraints.

Starting from an initial solution $s_0 \in \mathbf{sol}(O)$, a *Local Search Algorithm* iteratively navigates the space $\mathbf{sol}(O)$ by stepping from one solution s to a neighbor $s' \in \mathcal{N}(s)$, obtained using a move m. We write $s' = s \oplus m$ to denote the application of the move m to the solution s, obtaining the solution s'. This process is illustrated in figure 3.1. The selection of m is based on the values of f on $\mathcal{N}(s)$ and it depends on the specific LS technique considered. It can also be the case that $f(s') = f(s)$ (*idle* or *sideway* move) or $f(s') > f(s)$ (*worsening* move).

The *stop criterion* specifies when the iterations of the LS algorithm must be stopped, and the best solution found in the process is returned. The actual stop criterion depends on the technique at hand and it may be based on the following possibilities:

- the specific qualities of the solution reached (the objective function has reached a desired minimum value, or a value close to a known lower bound)
- a maximum number of iterations performed;
- a temporal timeout.

3.2 Basic Local Search Algorithms

A general procedure for an abstract Local Search Algorithm (see [24]) is shown in the following code:

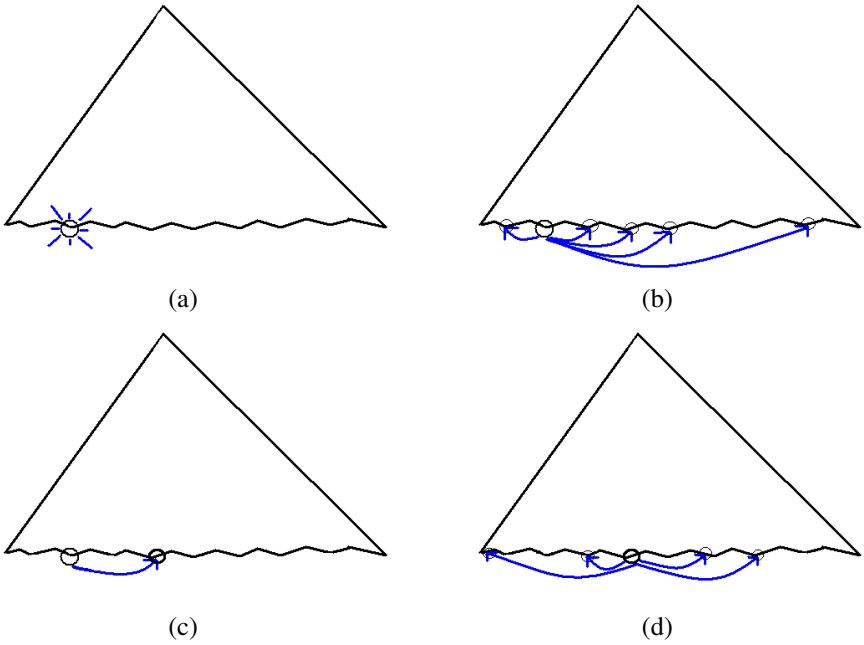


Figure 3.1: LS exploration on the search tree: (a) Initial solution s_0. (b) Exploration of the neighborhood of s_0. (c) Moving to a neighbor. (d) The process is iterated.

Component	**Hill Climbing**	**Simulated Annealing**	**Monte Carlo**	**Tabu Search**
InitialSolution	not specified	random	random	not specified
SelectMove	random/best/...	random	random	best non tabu
AcceptableMove	non-worsening	improving always; if worsening with probability $e^{-\Delta/T}$	improving always; if worsening with probability p	always
StopSearch	idle iterations	frozen system	idle iterations	idle iterations

Table 3.1: Differences between the Basics Local Search Techniques

```
(1)   procedure LS(SearchSpace S, Neighborhood N, CostFunc F);
(2)   {
(3)      s0 := InitialSolution(S);
(4)      i := 0;
(5)      while ( StopCriterion( si; i) ) do
(6)      {
(7)         m := SelectMove( si; F; N );
(8)         if ( AcceptableMove( m; si; F ) )
(9)            then si+1 := si ⊕ m;
(10)        else si+1 := si;
(11)        i := i + 1
(12)     }
(13)  }
```

We assume that the CSP underlying the COP is consistent, so there always exists an initial solution s_0, which is a leaf of the search tree: this first solution can be randomly generated or freely constructed by some specific heuristic. The `StopCriterion` method defines when to stop the search and return the result. `SelectMove` choose a neighbor s' in the neighborhood $\mathcal{N}(s_i)$ and `AcceptableMove` determines if the solution s' can be accepted as the next one. Each of these characteristics may vary according to different strategies and leads to different LS techniques. In the following sections we describe the most common ones and table 3.1 shows the main characteristics of their traditional implementations.

3.2.1 Hill Climbing

Hill Climbing is the simplest LS technique and it is based on the following idea to accept the move: at each step of the search, the algorithm *accepts a move that either improves the value of the objective function or leave it unchanged*. There are several variants of the Hill Climbing technique (see for example the ones presented in [54, 63, 70]). They typically share the same definition of `AcceptableMove`, but have different move selection criterions (random

move, best move...) and stop criterions (number of iterations, number of idle iterations, good objective function value reached, ...).

The main problem with Hill Climbing procedures is that they usually fall into *local minima* (i.e. solutions whose neighborhood is made up of solutions having greater cost) and cannot go out of them (because they cannot accept worsening moves).

3.2.2 Simulated annealing

Simulated Annealing was proposed by Kirkpatrick et al. [45] and Cerny [76], and gets that name after an analogy with a simulated controlled cooling of a collection of hot vibrating atoms. The idea is based on accepting non-improving moves with probability that decreases with time. The process starts by creating a random initial solution s_0. At each iteration a neighbor s' of the current solution s is generated. We define $\Delta = f(s') - f(s)$ that measure the improvement given by the new candidate solution. If the neighbor improves the objective functions value ($\Delta \leq 0$), it is accepted; if it is a worsening solution ($\Delta > 0$), it is accepted with probability $e^{-\Delta/T}$, where T is a parameter, called the *temperature*. The temperature T is initially set to an appropriately high value T_0. After a fixed number of iterations, the temperature is decreased by the *cooling rate* α, so that at each cooling step n, $T_n = \alpha \times T_{n-1}$. In general α is in the range $0.8 < \alpha < 1$. The procedure stops when the temperature reaches a low-temperature region, i.e., when no solution that increases the cost function is accepted anymore. In this case we say that the system is frozen.

Note that in the initial stage of the search (i.e., at "high temperatures"), Simulated Annealing does not fall into local minima, but for "low temperatures" the procedure reduces to an Hill Climbing, therefore the final state obtained by Simulated Annealing will clearly be a local minimum.

3.2.3 Monte Carlo

The Monte Carlo method [30, 51] is similar to Simulated Annealing. It only differs on the probability to accept a worsening solution. Instead of a probability decreasing with time, the Monte Carlo method has a fixed probability to accept a worsening solution, while it always accepts improving moves (as Simulated Annealing does). In this way, the system never gets frozen and it is theoretically possible to reach every possible state of the problem (Monte Carlo method belongs to the family of ergodic methods). The name Monte Carlo is a reference to the Monte Carlo Casino in Monaco. Von Neumann chose this name for the stochastic method he was applying to investigate about radiation shielding, while working at the Los Alamos Scientific Laboratory with Stanislaw Ulam [29, 50].

3.2.4 Tabu Search

Tabu Search is a method in which a fundamental role is played by keeping track of features of previously visited solutions. It was proposed by Glover [35, 36, 37] in the late 1980s, and since then it has been employed by many researchers for solving problems in different domains. The basic mechanism of Tabu Search is quite simple: at each iteration, given the current solution s, a subset $Q \subseteq \mathcal{N}(s)$ of its neighborhood is explored. *The solution*

$s' \in Q$ that gives the minimum value of the cost function becomes the new current solution independently of the fact that its value is better or worse than the value of s. To prevent cycling, there is a so called *tabu list*, i.e., a list of moves that are forbidden to be performed. The tabu list comprises the last k moves (k is a parameter of the method), and it is run as a queue (after k iteration into the tabu list, the move is discarded and can be selected again). The prohibition mechanism of the tabu list can be implemented in many ways, leading to several tabu search variants. An interesting improvement of the basic idea is the one proposed by Gendreau et al. [33]: each performed move is inserted in the tabu list together with the number of iterations *tabu_iter* that it is going to be kept in the list. The number *tabu_iter* is randomly selected between two given parameters *tmin* and *tmax* (with $tmin \leq tmax$). Each time a new move is inserted in the list, the value *tabu_iter* of all the moves in the list is updated (i.e., decremented), and when it gets to 0, the move is removed.

The main characteristic of tabu search (that differentiates it from Hill Climbing or Simulated Annealing) is that tabu search *avoids the local minima*: in fact a new $s' \in \mathcal{N}(s)$ is always chosen, even if it is worse than the current solution.

3.3 Advanced Local Search Techniques

A lot of different improvements can be developed starting from the basic LS algorithms described in the previous section. Here we summarize the main ones (cf. [24, 37] for a complete analysis).

- *Diversification strategies*: the main problem of LS techniques is that they usually fall into local minima. So LS algorithms may use some form of diversification mechanism that allows the search to escape from local minima and to move far away from it, to unexplored portion of the search space that may contain better solutions. The Tabu list as well as the stochastic acceptation mechanism of Simulated Annealing implement two diversification strategies: both these approaches try to escape from local minima accepting worsening moves. The problem is that once a good solution has been found, it is reasonable to intensify the search in its proximity to find a better one. So it is necessary to find a good balance between two conflicting objectives: a good strategy should both *diversify* and *intensify* by moving outside the attraction area of already visited local minima, but not too far from it.

- *Learning approach*: one common aspect of LS technique like Hill Climbing and Simulated Annealing is that they are totally *oblivious*, i.e., they do not have any memory about the history of the search. Even if Tabu Search has a elementary memory mechanism that avoids to perform again recent moves, several more sophisticated learning approaches can be implemented. For example, it could be possible to learn which variables give a bigger contribution to the objective function and focus the search on these variables. Moreover parameters can be tuned at run-time (e.g., the length of the tabu list, the cooling rate, ...), adapting the search to the portion of the search space we are analyzing. A commonly used learning approach is the Reactive Tabu Search (see for example [4, 8, 11]): it is a Tabu Search algorithm that monitors the occurrence of cycles and their repetition and reacts to them by adapting the Tabu list size.

- *Composition*: an interesting property of the Local Search paradigm is that different techniques can be combined and alternated to give rise to complex algorithms. Different LS strategies can be alternated on the same search space, or different neighborhood relation can be defined, in order to explore different kind of neighborhood according to different LS strategies or different moments of the search process. These ideas has been classified and systematized in [24] under the name of *Multi Neighborhood Search*: a set of operators for combining neighborhood has been defined as well as basic composite LS techniques, based on different neighborhoods.

3.4 Other Meta-heuristics

All the methods presented so far belong to the class of *trajectory methods*: they all work on a single solution at any time, describing a trajectory in the search space during the search process. Also non-trajectory methods have been studied in literature, with the name of *population-based* approaches. These methods do not work on a single solution, but manipulate a set of solutions at the same time. Some of them have been successfully applied to a wide variety of real-life and large optimization problems. Moreover, also combination of different meta-heuristics have been proposed, under the name of *hybrid meta-heuristic*. A deep analysis about these meta-heuristics is out of the scope of this thesis, but in this section we give a brief outline of these methods.

3.4.1 Population-based methods

Population-based methods are iterative meta-heuristics that deal with a set of solution, rather than with a single solution. In this context, a single solution is called *individual* and the set of individuals is called *population*. All the individuals cooperate together, driven by the specific algorithm, in order to find the good solution to the problem considered. The most studied population-based methods are Evolutionary Computation (EC) and Ant Colony Optimization (ACO), described below. We suggest to look at [10] for a deep analysis.

- *Evolutionary Computation*: these methods are also know as *genetic algorithms*, and use mechanisms inspired by biological evolution, such as reproduction, recombination, mutation and selection. The algorithm usually start *generating a population*, i.e., creating, randomly or heuristically, a large number of different individuals (i.e., solutions). Each step starts with a *reproduction* phase, where couples or small groups of individuals combine together, generating new individuals. The new individuals are generated by a *recombination* of the parent individuals, that exchange portions of their solutions, for example in a crossover fashion. Then every single individual may be subject to a *mutation*: i.e., it can stochastically change part of its solution. At this point there is the last phase, called *selection*. According to the objective function (in this context called *fitness function*), all the individuals are evaluated, and only the good ones can survive, while the other ones are discarded. Then the process iterates from the reproduction phase. After a large number of iterations the average quality of the individuals increases, leading to good individuals, representing good solutions. There

are several strategies that can be applied to this basic schema, leading to a variety of different genetic algorithms.

- *Ant Colony Optimization*: it is a probabilistic technique that can be used to find good paths through graphs. It has been firstly applied to the TSP problem, it can be easily extended to solve computational problems which can be reduced to TSP. The original idea comes from observing the exploitation of food resources among ants. In an ant colony, after an ant has discovered the food, it comes back to the nest, following any route. Every ant leave on its path a trial of *pheromone*, a chemical factor that is recognized by the other ants and gives them a *positive feedback*. When the first ant arrives to the nest, other ants start a path to reach the food, in part randomly, in part driven by the pheromone. It is important to note that pheromone *evaporates* over time: so if the chemical factor is not enforced by other ants passing through the same way and leaving more pheromone, it disappears. The ants keep going to the food source and back to the nest. Ants pass through the longest path less frequently, because it requires more time to be crossed and the pheromone is not enough enforced. Thus, the amount of pheromone on that path decreases. On the other hand, the shorter route will be traveled by more ants than the long one. Thus, the short route will be increasingly enhanced, and therefore become more attractive. The long route will eventually disappear because pheromones are volatile. In this way all the ants have determined and therefore chosen the shortest route. An ACO algorithm simulate the behavior of the ants (choosing a route, randomly following the pheromone) and of the pheromone (evaporating over time), allowing to discover the shortest path on a graph.

3.4.2 Hybrid meta-heuristics

When dealing with large-scale real-life problems the use of a single meta-heuristic can be not effective enough to obtain satisfying results. Indeed, a skillful combination of concepts from different meta-heuristics can lead to more efficient and flexible behaviors. Thus, it can be worthy to incorporate typical operation research techniques, e.g., dynamic or integer or linear programming techniques, together with meta-heuristics approaches. Moreover, approaches coming from artificial intelligence, such as multi-agent systems, data mining and so on, can be very fruitful. Combinations of meta-heuristics with other meta-heuristics or optimization strategies are called *hybrid meta-heuristics*.

Designing and implementing hybrid meta-heuristics requires to consider several problems that do not occur when dealing with a single meta-heuristic. In particular the *choice and tuning of the parameters* became crucial: every component has its own parameters that not only affects its behavior but also the other components and the overall algorithms; moreover, the combinations of parameters values grow exponentially. Another important aspect to consider is the *separation and interactions between the components*: it may take place at low level or high level, components may share the same data structures or communicate via messages, the components may cooperate at the same level or act in a master-slave fashion. Finally, it is very difficult to theoretically demonstrate properties of hybrid meta-heuristic, because of the complex interaction between components. Thus, research on this field is mostly based on *experimental work*: the proper design of experiments and the validity of analysis of experi-

mental work for this kind of algorithms is a key aspect to take in strong consideration. In this thesis we develop a tool for developing a hybrid meta-heuristic integrating Constraint Programming and Local Search techniques. For a general discussion on hybrid meta-heuristics, we refer the interested reader to [9].

3.5 Examples

In this section we use the SCTT problem to show in practice some concepts explained in this chapter. A possible move for this problem can be defined by a couple of variables belonging to the same course, i.e., $m = (x_{c,i}, x_{c,j})$, with $c \in C$ and $i, j \in P$. The application of the move is performed by swapping the values of the two variables involved. This kind of move is usually referred with the name of *swap move*. Formally, given a state s and a swap move $m = (x_i, x_j)$, the state $s' = s \oplus m$ is obtained as follows: for each x_k with $k \neq i, k \neq j$, we have $x_{k_{s'}} = x_{k_s}$; for i and j we have $x_{i_{s'}} = x_{j_s}$ and $x_{j_{s'}} = x_{i_s}$. Note that in our example the swap move does not affect the constraint C1, since it does not add or remove lectures from a row. Thus, if we apply a swap move to a solution that satisfies C1, we obtain another solution that still satisfies C1.

Let us consider a Tabu Search algorithm that works on swapping moves, having a FIFO tabu list that can contain at most 3 elements. The algorithm starts with the empty tabu list [] and the following initial solution s_0:

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	1	1	0	0	1	0	0	0	0	0	3 − 2 = 1
PL	0	0	0	0	0	0	1	1	1	1	4 − 2 = 2
AI	1	1	1	0	0	0	0	0	0	0	3 − 2 = 1
s_0							Total FObj:				4

We now simulate some steps of the algorithm. The following random move is selected: $m_1 = (x_{PL,TH_m}, x_{PL,WE_a})$. It can be applied to s_0, since its application does not lead to any constraint violation and it is an improving move. Thus, we obtain the following state $s_1 = s_0 \oplus m_1$.

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	1	1	0	0	1	0	0	0	0	0	3 − 2 = 1
PL	0	0	0	0	0	1	0	1	1	1	4 − 3 = 1
AI	1	1	1	0	0	0	0	0	0	0	3 − 2 = 1
s_1							Total FObj:				3

Now the tabu list is $[m_1]$ and the current solution s_1 is also the best known one. The next random move generated is $m_2 = (x_{OS,MO_m}, x_{OS,TH_m})$. It can be applied, since it does not violated any constraint, it is not equal to any move in the tabu list and it is an improving move. Its application leads to state $s_2 = s_1 \oplus m_2$, i.e, the following :

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	0	1	0	0	1	0	1	0	0	0	$3 - 3 = 0$
PL	0	0	0	0	0	1	0	1	1	1	$4 - 3 = 1$
AI	1	1	1	0	0	0	0	0	0	0	$3 - 2 = 1$
s_2							Total FObj:				2

Now the tabu list is $[m_2, m_1]$ and the current/best solution is s_2. Let us suppose that next random move generated is $m_3 = (x_{OS,MO_m}, x_{OS,TH_m})$. It does not violate any constraint, so it could be applied. But it is a worsening move and Tabu Search accept a worsening move only if it is not equal to any move in the tabu list. In this case we have m_2 in the tabu list, and $m_3 = m_2$. So, the prohibition mechanism of the Tabu Search forbids the application of move m_3, thus avoiding cycling between states. Now the move $m_4 = (x_{PL,MO_m}, x_{PL,WE_a})$ is generated. It can be applied because it does not violate any constraint. It is an idle move, since the objective function values would remain unchanged after its application. All the local search algorithms usually apply idle moves: even if these moves do not improve the objective function, they modify the solution leading to a different state, and this can help escaping from local minima. So, m_4 is applied, the tabu list now contains $[m_4, m_2, m_1]$ and the current/best state is $s_3 = s_2 \oplus m_4$, defined as follows:

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	0	1	0	0	1	0	1	0	0	0	$3 - 3 = 0$
PL	1	0	0	0	0	0	0	1	1	1	$4 - 3 = 1$
AI	1	1	1	0	0	0	0	0	0	0	$3 - 2 = 1$
s_3							Total FObj:				2

At this point the move $m_5 = (x_{PL,WE_a}, x_{PL,FR_a})$ is generated. It does not violated constraints and it improves the objective function, thus it is accepted. The following state is obtained, i.e., $s_4 = s_3 \oplus m_5$.

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	0	1	0	0	1	0	1	0	0	0	$3 - 3 = 0$
PL	1	0	0	0	0	1	0	1	1	0	$4 - 4 = 0$
AI	1	1	1	0	0	0	0	0	0	0	$3 - 2 = 1$
s_4							Total FObj:				1

Now the tabu list is $[m_5, m_4, m_2]$, since the maximum length of the tabu list is 3. State s_4 is both the best and the current state. Suppose that the move $m_6 = (x_{PL,TH_a}, x_{PL,FR_a})$ is generated: it does not violated constraints, but it is a worsening move. Since there is no move in the tabu list equal to m_6, the move is applied anyway. Thus, the best state so far is still s_4, but the current state for the search process became $s_5 = s_4 \oplus m_6$, i.e.:

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	0	1	0	0	1	0	1	0	0	0	$3 - 3 = 0$
PL	1	0	0	0	0	1	0	0	1	1	$4 - 3 = 1$
AI	1	1	1	0	0	0	0	0	0	0	$3 - 2 = 1$
s_5							Total FObj:				2

The next move generated is $m_7 = (x_{PL,TH_a}, x_{PL,FR_m})$. It is applied and it leads to the state $s_6 = s_5 \oplus m_7$, represented as follows.

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	0	1	0	0	1	0	1	0	0	0	$3 - 3 = 0$
PL	1	0	0	0	0	1	0	1	0	1	$4 - 3 = 0$
AI	1	1	1	0	0	0	0	0	0	0	$3 - 2 = 1$
s_6							Total FObj:				1

Then, suppose that the following move is generated, $m_8 = (x_{AI,MO_a}, x_{AI,WE_m})$. It can be applied and leads to the state $s_7 = s_6 \oplus m_8$, represented as follows.

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	0	1	0	0	1	0	1	0	0	0	$3 - 3 = 0$
PL	1	0	0	0	0	1	0	1	0	1	$4 - 3 = 0$
AI	1	0	1	0	1	0	0	0	0	0	$3 - 3 = 0$
s_7							Total FObj:				0

The objective function value for state s_7 is better than the one of the best state obtained so far, i.e., s_4. So, s_7 became both the best and the current state. The 0 value is a lower bound for this problem, so state s_7 is an optimum solution. The Tabu Search algorithms continues but it can only apply idle moves, worsening moves, or improving moves subsequent to worsening moves, thus not improving the best known solution. After a certain amount of idle iterations, the search will stop.

4

Hybridization of CP and LS

We can split the methods used to solve CSPs and COPs into two main categories: complete methods and incomplete methods (see [60]). *Complete methods* systematically explore the whole search space, looking for a solution (for CSPs) or an optimal (for COPs) one. *Incomplete methods* rely on heuristics that focus only on some areas of the search space to find a solution (CSPs) or a non-proved to be optimal one (COPs).

Constraint Programming and Local Search belong respectively to the categories of complete and incomplete methods and have opposite characteristics. The main advantage of CP is *flexibility*: this programming paradigm allow to model problem in a declarative way, obtaining compact and flexible encodings, where adding new constraints or new requests is straightforward, and it does not affect the model previous model. Moreover it is a complete method, so it can exhaustively search the solutions space and if a problem is small enough, an optimal solution can be found in reasonable time. Theoretically, even with large problems the optimal solution can be found by a constraint programming algorithm, as long as we have infinite time to wait the process to analyze the whole possible solutions of the problem. On the other hand, the strong point of LS (and in general of incomplete methods) is *efficiency*: the algorithms based on this paradigm do not analyze the whole search space, but focus the search only on interesting and promising areas, so they can reach a good solution (without any warranty of optimality) in very short time, even if the problem is very hard.

In the past decades people started thinking about hybridizing complete and incomplete methods (and in particular, CP and LS) to obtains reliable and efficient algorithms that may inherit advantages of both these different paradigms. In the next section (4.1) we illustrate the hybrid schemas commonly found in literature. Section 4.2 focuses on Large Neighborhood Search, a special LS technique that is typically implemented in a hybrid fashion and that has been deeply used in this thesis. Section 4.3 introduces some new hybrid schemas, and the last section (4.4) present the language Comet, a state-of-the-art language for hybrid methods in combinatorial optimization.

4.1 Consolidated Hybrid Techniques

Two major types of approaches to combine the abilities of Constraint Programming and Local Search are presented in the literature [31, 44]. These hybrid methods follow the *master-*

slave paradigm, i.e., one method is the master algorithm and the other one act as the slave, performing prefixed tasks during the search process. We can summarize these approaches as follows:

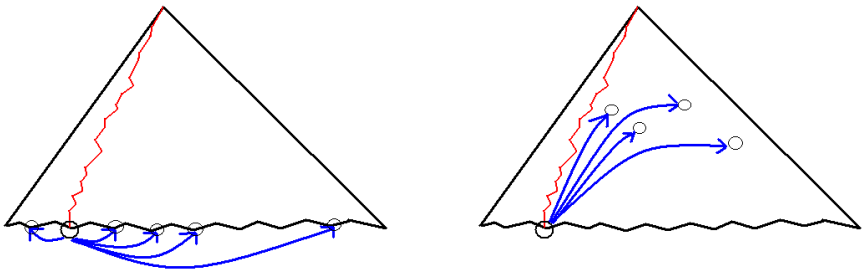
1. *CP master*: a systematic-search algorithm based on constraint programming can be improved by inserting a local search algorithm at some point of the search procedure, e.g.:
   (a) at a leaf (i.e., on complete assignments or on an internal node (i.e., on a partial assignment) of the search tree explored by the constraint programming procedure, in order to improve the solution found;
   (b) at a node of the search tree, to restrict the list of child-nodes to explore;
   (c) to generate in a greedy way a path in the search tree;
2. *LS master*: a local search algorithm can benefit of the support of constraint programming, e.g.:
   (a) to analyze the neighborhood and discarding the neighboring solution that do not satisfy the constraints;
   (b) to explore a fragment of the neighborhood of the current solution;
   (c) to define the search of the best neighboring solution as a problem of constrained optimization (COP).

In the experiment performed in this thesis we adopted the two hybrid techniques 1(a) and 2(a-b) and we apply them to several problems. In the first approach we employ constraint programming for searching an initial solution and subsequently we improve it by means of local search, using classical algorithms like hill climbing, steepest descent and tabu search (see representation in figure 4.1). In the second approach we improve the initial solution devising Large Neighborhood Search algorithms that exploits a constraint programming model for the exploration of large neighborhood fragments.

4.2 Large Neighborhood Search

Large Neighborhood Search (LNS), first introduced in [64] is a LS method that relies on a particular definition of the neighborhood relation and of the strategy to explore the neighborhood. Differently from traditional LS methods, where the existing solution is modified just by small changes to a limited number of variables (as is typical with LS move operators), LNS selects a large subset of the problem variables and explores it for improving solutions (that's why it is called "Large" Neighborhood Search). The subset of the problem can be represented by a set of variables $FV \subseteq \mathcal{X}$, that we call *free variables*. Defining FV corresponds to define a neighborhood relation.

According to the classification given in the previous section, LNS is a LS strategy that can be naturally implemented using CP, leading to hybrid algorithms that involves the approaches 2(a), 2(b), and 2(c) of the above classification: CP can manage and exhaustively explore a



(a) (b)

Figure 4.1: A step of LS (a) and LNS (b) applied to a initial solution obtained by CP

lot of variables subjected to constraints, so it is a suitable method for the exploration of large neighborhoods.

Three aspects are crucial in LNS definition, w.r.t. the performance of this technique: *(1) which and (2) how many variables* have to be selected (i.e., the definition of FV), and *(3) how to perform the exploration* on these variables. Let us briefly analyze these three key-points, starting from the third one.

(3) Given FV, the *exploration* can be performed with any searching technique: CP, Operation Research algorithms, and so on. We can be interested in different kind of search: we may want to find the best neighborhood or the best neighborhood within a certain exploration timeout; me may also try a more lazy approach, for example searching for the first improving neighborhood or the first neighborhood improving the objective function of at least a given value, and so on. (2) Deciding *how many variables* will be free (i.e., $|FV|$) affects the time spent on every large neighborhood exploration and the improvement of the objective function for each exploration. A small FV will lead to very efficient and fast search, but with very little improvement of the objective function. Otherwise, a big FV can lead to big improvement at each step, but every single exploration can take a lot of time. This trade-off should be investigated experimentally, looking at a dimension of FV that leads to fast enough explorations and to good improvements. Obviously, the choice of $|FV|$ is strictly related to the search technique chosen (e.g., a strong technique can manage more variables than a naïve one) and to the use or not of a timeout. (1) The choice of *which variables* will be included in FV is strictly related to the problem we are solving: for simple and not too structured problems we can select the variables in a naïve way (randomly, or iterating between given sets of them); for complex and well-structured problems, we should define FV cleverly, selecting the variables which are most likely to give an improvement to the solution.

Considering the SCTT example from the previous chapter, a LNS algorithm may define a large neighborhood selecting two days and considering free all the variables belonging to the selected days. Thus, the set FV of free variables corresponding to the couple of days $(Monday, Tuesday)$ sets all the variable of the first four columns free. Figure 4.2 shows these free variables with the original domains, ready to be explored by a search engine.

Periods →	MO		TU		WE		TH		FR		FObj
Courses ↓	m	a	m	a	m	a	m	a	m	a	$\delta(c) - \#days$
OS	01	01	01	01	1	0	1	0	0	0	undefined
PL	01	01	01	01	0	1	0	1	1	1	undefined
AI	01	01	01	01	1	0	0	0	0	0	undefined
							Total FObj:				1

Figure 4.2: LNS move for a solution of SCTT

4.3 New Hybrid Techniques

Monfroy, Saubion e Lambert in [55] proposed a new paradigm that hybridizes Constraint Programming and Local Search without using the traditional master-slave schema. The new approach consists in splitting CP and LS into several basic components that can be managed and activated at the same level: three *basic operations* are defined, i.e., (1) constraint propagation, (2) LS moves, (3) splitting functions; these operations are considered at the same level and applied to unions of CSPs. The execution order (alternated, consecutive...) of the functions is totally free, so this framework makes it possible to design smarter strategies than traditional master-slave paradigms. In [46, 55] they applied these ideas on a uniform generic hybridization framework based on K.R. Apt's chaotic iterations: the declared aim of the framework is not to provide competitive results, nor to propose specific problems driven combinations to compete with state of the art solvers, but to be an easy tool to study more precisely the benefit of hybridization and to design new hybridization strategies.

Pascal Van Hentenryck and Laurent Michel proposed in [52, 74] the programming language Comet [75], where constraint programming, mathematical programming and local search are integrated. Since Comet has become a state-of-the-art tool for Large Neighborhood Search, in the next section we give a short overview of this language.

4.4 The Comet language

Comet supports both modeling and search abstractions, and uses constraint programming to describe and control LS, in an environment where the different paradigms share the same search primitives. One of the key characteristics of the Comet system is *constraint-based local search* (CBLS), i.e., the idea of specifying local search algorithms as two components: (1) a high-level *model* describing the applications in terms of constraints, constraint combinators, and objective functions; (2) a clearly separated *search procedure* that implicitly takes advantage of the model in a general fashion. The computational model underlying constraint-based local search uses constraints and objectives to drive the search procedure toward feasible, high-quality solutions. Comet also allows parallel and distributed computing, providing primitives to manage threads, processes, and synchronization. Since 2009 Comet language is the main software product of the Dynamic Decision Technologies Inc., a company with headquarters in Providence (RI, USA) and Louvain-La-Neuve (Belgium) that provides solutions and technology for optimization problems. Thus, Comet is not open source, and it is

free downloadable only for academic purposes, with some restrictions.

Comet is a full object-oriented programming language that provides garbage collection and features some advanced control structures for search and parallel programming. It provides basic data types, such as integer, floats, booleans and strings, as well as complex ones, such as ranges, arrays and matrices, set, queues and so on. Standard control constructs, like if-then-else, switch, for, forall, and do-while are available. It is possible to use selectors, i.e., functions that allow to select randomly or according to a probability or an evaluation function an element in a set or a range. Standard functionalities to define functions, classes and interfaces are also available, together with mechanisms to include files and libraries. Comet also comes with the support for threads and events, allowing to exploit parallelism.

Together with these standard programming language features, Comet allows to define CSPs and COPs. It has CP primitives to declare variables, domains, constraints and objective functions. It also allows the user to implement his/her own propagators, i.e. an algorithms associated to a constraints, in charge of removing inconsistent values from the domain of the variables. Then Comet provides several primitives that allows the user to define the search strategy to use among a model, i.e., how to choose the variables, how to slipt the problem into subproblems, how to navigate the search space, what action to perform when a solution or inconsistency is reached, and so on. Also a set of LS primitives to define neighborhoods and transition between neighborhoods are provided. In particular, the LS approach in Comet exploits the use of invariants i.e., one-way declarative constraints that specify a relation which must be maintained under assignments of new values to the participating variables. Finally, Comet provides mathematical programming functionalities, such as the simplex algorithm and column generations procedures.

In [5] Comet has been applied to a Vehicle Routing Problem with Time Windows. The algorithm first minimizes the number of vehicles using simulated annealing. It then minimizes travel cost using a large neighborhood search which may relocate a large number of customers. Experimental results demonstrate the effectiveness of the algorithm which has improved 10 (17%) of the 58 best published solutions to the Solomon benchmarks [65], while matching the best solutions in 46 benchmarks (82%). Comet ha also been applied in [73] to several online stochastic reservation problems, i.e., problems where requests come online and must be dynamically allocated to limited resources in order to maximize profit. Multi-knapsack problems with or without overbooking are examples of such online stochastic reservations. This work presents a constant sub-optimality approximation of multi-knapsack problems and demonstrates the effectiveness of the regret algorithm on multi-knapsack problems (with and without overbooking) based on the benchmarks proposed earlier. In [53] Comet uses parallelism transparently to speed up constraint programs. This paper shows how to parallelize constraint programs transparently without changes to the sequential code, in order to exploit the availability of commodity multi-core and multi-processor machines. Experimental results show that the parallel implementation may produce significant speedups.

II

Software Tools

5

Gecode

Gecode (**Ge**neric **co**nstraint **d**evelopment **e**nvironment) [67] is a programming environment for developing constraint-based systems and applications. It is a C++ state-of-the-art constraint programming toolkit, modular and extensible. Its development started in 2002 by a team composed by Christian Schulte (head of the project), Mikael Lagerkvist and Guido Tack, that keeps improving the project and adding functionalities: since the first 1.0.0 release (December 2005) to now (December 2010) around 24 progressive new versions has been released (the last one is the 3.4.0), with a mean of a new version every 2.5 months. Moreover, its users community is constantly growing.

In this thesis we use Gecode for our constraint programming purposes, combining it with EasyLocal++, a Local Search framework that will be introduced in chapter 6. In this chapter we start explaining the main characteristics of Gecode (section 5.1), and then we give details about its architecture (section 5.2). In the last section (5.3) we show how to model a problem using Gecode.

5.1 Gecode Overview

Gecode has many characteristics that helped it to became a standard constraint programming framework toward the computer science community.

First of all it is *open source*: it is free, distributed under the MIT license and listed as free software by the Free Software Foundation [66]. All of its part (source code, example, documentation...) are all available for download. This also means that anyone can modify the implementation of some classes, extend or improve its functionalities, interface it to other systems in an easy way.

Gecode is *portable*: it is completely implemented in C++, carefully following the C++ standard. It can be compiled with modern C++ compilers and runs on a wide range of machines (Windows, Mac, Linux, Unix, CygWin...), including also 64bit machines.

It is deeply *documented*, since user are provided with: (1) a tutorial [69] that explain the Gecode architecture and how to program in Gecode (with several examples and use cases); (2) an on-line reference documentation [68] that gives the technical specifications of each module/class/function implemented in the framework; (3) a user-mailing-list where Gecode programmer and users may share opinions and implementation issues, report bugs,

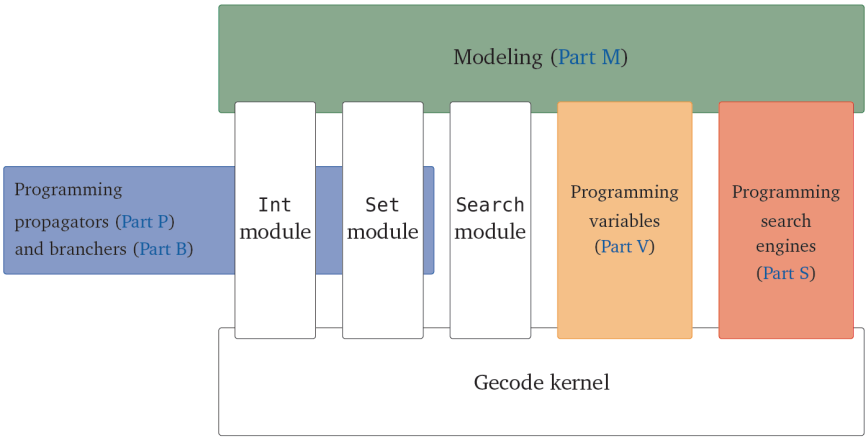


Figure 5.1: Gecode Architecture

suggest solutions and so on.

Gecode has excellent *performance* with respect to both runtime and memory usage. It won severals competitions, such as the MiniZinc Challenge, both 2009 and 2008.

Gecode also implements *parallel* search: it exploits the multiple cores of today's hardware for parallel search, giving an already efficient base system an additional edge.

5.2 Gecode Architecture

Gecode environment is made of 5 main modules, spread into 3 layers, as shown in figure 5.1: the *kernel* module, on the bottom of the architecture; the *finite domain*, the *set domain* and the *search* modules, that constitute the middle layer; the *modeling* module, that is the top layer.

Kernel Gecode's kernel implements common functionalities (such as data structures, basic operations...) upon which the modules for integer and set constraints as well as search engines are built. It provides a comprehensive programming interface to construct new variable domains, *propagators*, *branchers* and *search engines*. Propagators are implementations of constraints that perform consistency checking and propagate constraint information, in order to narrow the domains. Branchers are modules that perform variable assignment, typically applying domain splitting rules. The kernel is slim (around 1700 lines of code) and requires no modification or hacking for adding new variable domains or search engines.

Finite Domain The Finite Domain module implements finite domain constraints on top of the generic kernel. It offers standard constraints such as arithmetics, Boolean, linear

equations, and global constraints: distinct (alldifferent, with both bounds and domain consistency), count (global cardinality, with both bounds and domain consistency), element, scheduling (unary and cumulative resources including edge-finding and not-first, not-last propagation), table and regular constraints, sorted, sequence, circuit, channel, and so on. It is also possible to add new constraint and branchers to this module.

Set This module provides finite integer set variables. The standard set relations and operations are available as constraints, plus some specialized constraints, such as convexity, global reasoning for distinctness of sets, selection constraints, weighted sets, and constraints connecting finite domain and finite set variables. As for the finite domain constraints, the library can be easily extended with new constraints and branchers.

Search Search in Gecode can be performed using three different search engines. For the search on CSPs, a depth first search engine (called DFS) is provided. To deal with COPs a branch and bound (called BAB) engine and a depth first search with restart (called Restart) engine are provided. They act as described in section 2.3. It is also possible to implement new ad-hoc search engines. The search is based on a hybrid of *cloning* and *recomputation*. *Cloning* during search relies on the capability of a (partial) solution to create an exact copy of itself (needed when backtracking, while exploring alternatives of the solutions tree); *recomputation* remembers what has happened during the search space exploration: rather than storing an entire clone of a space, just enough information to redo the effect of exploration is stored.

The latest Gecode releases also provide advanced techniques including parallel search (that utilizes today's multi-core architectures), adaptive search (that speeds up further search) and path recomputation (that reduces propagation during recomputation).

Modeling Gecode comes with extensive modeling support, that allows the user to encode its problem using higher language facilities rather than rough low-level statements. The modeling support includes: regular expressions for extensional constraints; expressing aritmetic, set, and Boolean constraints in the standard way as expressions build from numbers, variables, and operators; also a matrix modeling support is provided.

5.3 Modeling with Gecode

In the Gecode philosophy, a model is implemented using *spaces*. A space is the repository for variables, constraints, objective function, searching options. Being C++ an object-oriented language, the modeling approach of Gecode exploits the *inheritance*: a model must implement the class *Space*, and the subclass constructor implements the actual model. In addition to the constructor, a model must implement some other functions (e.g., performing a copy of the space, returning the objective function, ...). A Gecode space can be asked to perform the propagation of the constraints, to find the first solution (or the next one) exploring the search tree, or to find the best solution in the whole search space.

Let us now focus on an example showing a Gecode model for the SCTT problem.

5.3.1 Model Head

Every Gecode model is defined into a class that inherits from a Gecode Space superclass. The superclasses available are `Space` for CSPs and `MinimizeSpace` and `MaximizeSpace` for minimization and maximization COPs. In the case of SCTT we declare the class `Timetabling` subclass of `MinimizeSpace` (line 1).

Line 2 declares the array `x` for the variables X of the problem, and line 3 declares the variable `fobj` that will contain the value of the objective function. `IntVarArray` and `IntVar` are built-in Gecode data structures. Line 4 starts the constructor method. It takes as input an object of the class `Faculty`, that contains all the information specific to the instance. All these input information are stored in the variable `in`. Line 5 sets variables and domains: array `x` is declared to contain a number of variables equal to `in.Periods()*in.Courses()`, all with a $\{0, 1\}$ domain. In line 6, the domain of variable `fobj` is set to take the range between 0 and the largest allowed integer value.

```
(1)    class Timetabling :  MinimizeSpace {
(2)       IntVarArray x;
(3)       IntVar fobj;
(4)       Timetabling(const Faculty& in ):
(5)          x( *this, in.Courses() * in.Periods() , 0, 1 ),
(6)          fobj( *this, 0, Int::Limits::max )
(7)       { ...
```

5.3.2 Constraints

Gecode provides a `Matrix` support class for accessing an array as a two dimensional matrix, since many models (like the one for SCTT) are naturally expressed using matrices. Line 9 set up a matrix interface to access `x`, with the number of columns equal to the number of time periods and the number of rows equal to the number of courses. The matrix has the same structure of the one in figure 1.1.

Then, lines 12–13 post constraint C1 on each row of the matrix. The constraint `linear` is a built in Gecode, and line 12 corresponds to the formula

$$\sum_{col} mat.row(r)[col] = in.NumberOfLectures(r)$$

where `in.NumberOfLectures(r)` contains the value of function l (number of weekly lectures) for each course (i.e., for each row).

Lines 15–19 post constraint C2. Line 18 corresponds to the constraint $x(c, r1) = 0 \vee x(c, r2) = 0$, applied to the variables of the same time periods sharing the same teacher.

Lines 20–23 post constraint C3. Line 23 corresponds to the constraint $x(period, c) = 0$, where `x(period,c)` corresponds to a time slot where the relative professor is unavailable.

```
(8)       unsigned int cols = in.Periods() ;
(9)       unsigned int rows = in.Courses();
(10)      Matrix<IntVarArgs> mat( x, cols, rows );
(11)
(12)      for ( int r = 0; r < rows; r++)
(13)         linear( *this, mat.row(r), IRT_EQ, in.NumberOfLectures( r ) );
(14)
(15)      for( int c = 0; c < cols; c++)
(16)         for( int r1 = 0; r1 < rows - 1; r1++ )
(17)            for( int r2 = r1+1; r2 < rows; r2++ )
(18)               if ( in.SameTeacher(r1,r2) )
(19)                  post(*this, tt( ( mat(c,r1)==0 ) || (mat(c,r2)==0) ) );
(20)
(21)      for( int period = 0; period < in.Periods(); period++)
(22)         for( int c = 0; c < in.Courses() ; c++ )
(23)            if ( !in.Available(c,period))
(24)               rel( *this, mat(period,c), IRT_EQ, 0 );
```

5.3.3 Objective Function and Branching

The formula for the objective function is constructed in lines 25–58. Note that some arrays of temporary variables are introduced in order to compute the intermediate calculus. They are declared with `IntVarArgs`, a Gecode built in data type for array of temporary variables). At the end of this piece of code (line 58), the formula is bound to the `fobj` variable.

Line 60 fixes the branching strategy. The variables to branch are the ones in the array `x`. The variable selection strategy is

```
tiebreak(INT_VAR_DEGREE_MAX,INT_VAR_SIZE_MAX)
```

i.e., the variable with the highest number of constraint on it is selected, breaking ties choosing the variable with largest domain size. The values selection strategy is `INT_VAL_MED`, i.e., the greatest value not greater than the median is selected.

Line 62 closes the constructor of the class `Timetabling`, opened at line 1.

```
(25)      IntVarArgs vectorWD( in.Courses() );
(26)      IntVarArgs sumWD( in.Courses() );
(27)      IntVarArgs diff( in.Courses() );
(28)      for( int course = 0 ; course < in.Courses(); course++ )
(29)      {
(30)         vectorWD[course].init(*this, 0, Int::Limits::max );
(31)         sumWD[course].init(*this, 0, Int::Limits::max );
(32)         diff[course].init(*this, Int::Limits::min, Int::Limits::max );
(33)      }
(34)      for( int course = 0 ; course < in.Courses(); course++ )
(35)      {
(36)         IntVarArgs workingDays( in.Days() );
(37)         for( int day = 0 ; day < in.Days(); day++ )
(38)         workingDays[day].init(*this, 0,1);
(39)
(40)         for( int day = 0 ; day < in.Days(); day++ )
(41)         {
(42)            IntVarArgs Day( in.PeriodsPerDay() );
(43)            for(unsigned int slot = 0 ; slot < in.PeriodsPerDay(); slot++ )
(44)            {
(45)               Day[slot].init(*this, 0,1);
(46)               int period = day*in.PeriodsPerDay()+ slot;
(47)               post(*this,tt( eqv(mat(period, course)!=0,Day[slot]==1)));
(48)            }
(49)            IntVar nLecturesPerDay(*this, 0, in.PeriodsPerDay());
(50)            linear(*this, Day, IRT_EQ, nLecturesPerDay );
(51)            post(*this,tt( eqv( nLecturesPerDay!=0,workingDays[day]==1)));
(52)         }
(53)         linear(*this, workingDays, IRT_EQ, sumWD[course]);
(54)         diff[course] = post(*this, in.CourseVector(course).MinWD() - sumWD[course]);
(55)         Gecode::max( *this, ZERO, diff[course], vectorWD[course] );
(56)      }
(57)
(58)      linear(*this, vectorWD, IRT_EQ, fobj );
(59)
(60)      branch(*this,x,tiebreak(INT_VAR_DEGREE_MAX,INT_VAR_SIZE_MAX),INT_VAL_MED);
(61)
(62)      }
```

5.3.4 Setting up a Branch and Bound search engine

In order to use the model defined above in a branch and bound search engine, a couple or more functions need to be defined into the `Timetabling` class. Function `cost`, that return the variable representing the cost of a solution, i.e., the objective function of the model. This function is defined in lines 63–65 and returns the variable `fobj`.

Moreover, a branch and bound search engine needs to know what constraint to add every time a new solution is found, in order to drive the search to better solutions and cut the search space. Function `constrain` (lines 67–71) defines the desired constraint to add: in line 67 it takes in input a `Space` object, (i.e., a solution of the model), and post a constraint (line 70) stating that the value of the variable `fobj` has to be less than the `fobj` value of the solution `sol`. The Gecode branch and bound search engine call this method, every time it find a new solution, passing the solution found as parameter.

```
(63)        IntVar Timetabling::cost(void) const {
(64)            return fobj;
(65)        }
(66)
(67)        void Timetabling::constrain(const Space& sol)
(68)        {
(69)            const Timetabling& s = static_cast<const Timetabling&>( sol );
(70)            rel(*this, fobj, IRT_LE, s.fobj.val() );
(71)        }
```

6

EasyLocal++

EasyLocal++ is an object-oriented framework that allows programmers to design, implement and test LS algorithms in an easy, fast and flexible way. The underlying idea is to capture the essential features of most LS meta-heuristics, and their possible compositions. This allows the user to address the design and implementation issues of new LS heuristics in a more principled way. EasyLocal++ has been entirely developed in C++ with wide use of object-oriented patterns (see [32]) and is the evolution of *Local++* (see [62]). The currently available version of EasyLocal++ is the 2.0 [28], which presents several refactorings w.r.t. the 1.0 version, published on Software and Practice [27]. Examples about how to use EasyLocal++ 1.0 can be found in [26].

In this thesis we used EasyLocal++ as our LS engines and extended its functionality to obtain a more general framework that combines together LS and CP. In section 6.1 the abstraction levels of EasyLocal++ are addressed and then its architecture is explained in further detail (section 6.2). Then section 6.3 shows how a COP can be encoded and solved using EasyLocal++.

6.1 EasyLocal++ Overview

The primary idea of EasyLocal++ is to capture the common aspects of local search meta-heuristics, and to provide the skeleton for general abstract LS algorithms. The framework thus provides the full control structures for the invariant part of the algorithms, and the user only supplies the problem specific details, e.g., defining concrete classes and implementing concrete methods. As common in frameworks, EasyLocal++ is characterized by the *inverse control* mechanism (also known as the *Hollywood Principle*: "Don't call us, we'll call you") for the communication between the general algorithm and the user code: the function of the framework calls the user-defined ones and not the other way round.

EasyLocal++ framework is composed by a *hierarchy of several layers of abstraction*, where each layer relies on the services supplied by lower levels and provides a set of more abstract operations. Figure 6.1 shows the different abstraction levels of EasyLocal++.

In particular we want to emphasize that the framework is split into two main parts: the *problem specific* (at the bottom) and the *problem independent* layers (at the top). The first two layers (i.e., *Data*, *Helpers*) are the *problem specific* ones: to define a COP, the user is

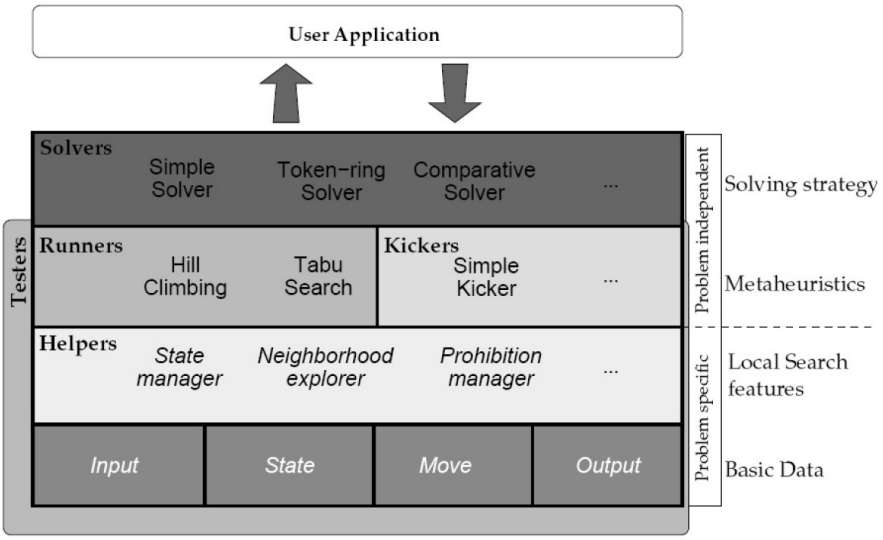


Figure 6.1: The levels of abstraction in EasyLocal++

required to implement the classes of these layers and nothing else. At this level the user defines characteristics of the COP to be solved, such as the problem model, the instances, the search space, the solutions as well as the transitions between solutions (i.e., the moves). The framework provides abstract classes, while the user defines his/her own concrete classes that inherits some characteristics from the abstract ones.

The next layers constitute the *problem independent* part of the framework: they define Local Search basic procedures (called *Runners*, as Hill Climbing, Tabu Search, Simulated Annealing...) and composite ones (called *Solvers* that combine together more runners). These layers operate on the data structures and definitions implemented at the lower levels. The user is not supposed to define anything at this point: he/she has only to *select* the meta-heuristic that the framework will use to solve the COP, along with search parameters (e.g., number of idle iteration before stopping, tabu list length, ...).

6.2 EasyLocal++ Architecture

In the following, we analyze in more detail each layer of the hierarchy.

Data The classes of this layer define the basic components for a LS algorithm. The class *Input* defines the characterization of an instance of the COP O we are solving, while the class *Output* defines how the solution found should be returned at the end of the search. The class *State* defines the characteristics of a solution in the search space (i.e., it describes a generic element $s \in \mathbf{sol}(O)$). The class *Move* defines the transition m between two solutions (*states* in the EasyLocal++ terminology): it doesn't specify how to perform the transition (i.e., the

operator $\oplus$, see section 3.1), but the information needed to perform it, such as the variables involved in the move.

These classes simply store attributes without any computation capabilities, which are instead provided by the Helper classes.

Helpers These classes collect the Local Search features, performing actions related to each specific aspect of the search. For example, the Neighborhood Explorer is responsible for everything regarding the neighborhood: selecting the move, updating the current state by executing a move (i.e., calculating $s' = s \oplus m$), and so on. Differing Neighborhood Explorers may be defined in case of composite search, each one handling a specific neighborhood relation used by the algorithm. The State Manager handles the attribute of each state, for example calculating the cost function, or establishing the feasibility of a state. Helpers work on data structures defined in the previous layer (especially instances of State and Move); they do not have their own internal data, but work on the internal states of runners and kickers, the next layer, that requires the helpers to perform task on their internal objects.

Runners These classes represent the algorithmic core of the framework. Their task is to perform a complete run of a Local Search algorithm, starting from an initial state and leading to a final one. Each runner has many data objects for representing the state of the search, such as the current state, the best state, the current move, number of iterations, and so on. A runner maintains links to all the helpers, which are invoked to perform problem-related tasks on the runner's own data. This way, runners can completely abstract from the problem description, and delegate problem-related tasks to the user-supplied classes that comply to a predefined helpers interface. Runners that encode the basic local search techniques, such as Hill Climbing, Simulated Annealing and Tabu Search are already defined into EasyLocal++.

Kickers A kicker is an algorithm that applies several moves to a current solution, with the aim to move away from the current state of the search, thus exploring new portions of the space of the solutions. Different Kicker classes implements different kinds of perturbations, such as a random kick, the best kick of a given length, etc. Kickers are usually combined together with runners. Typically at some point of the search, the execution of a runner is stopped and a kicker is executed, leading the search to a new current state, distant from the previous one. Subsequently the runner is reactivated, starting from the new state.

Solvers These classes represent the most abstract level, which is the external software layer of EasyLocal++. A solver controls the search by generating the initial solutions, and deciding how, and in which sequence, runners or kickers have to be activated. A solver can be made up by a single runner or by different runners and kickers in different combinations, leading to the construction of meta-heuristics or hybrid methods (called Multi Neighborhood Search strategies, see section 3.3). In addition, solvers communicate with the external environment, by getting the input and delivering the output. Runners and kickers as well as the Solvers are completely specified at the framework level. This means that in order to use them, the programmer has only to define an instance of the desired Solver class. New runners and

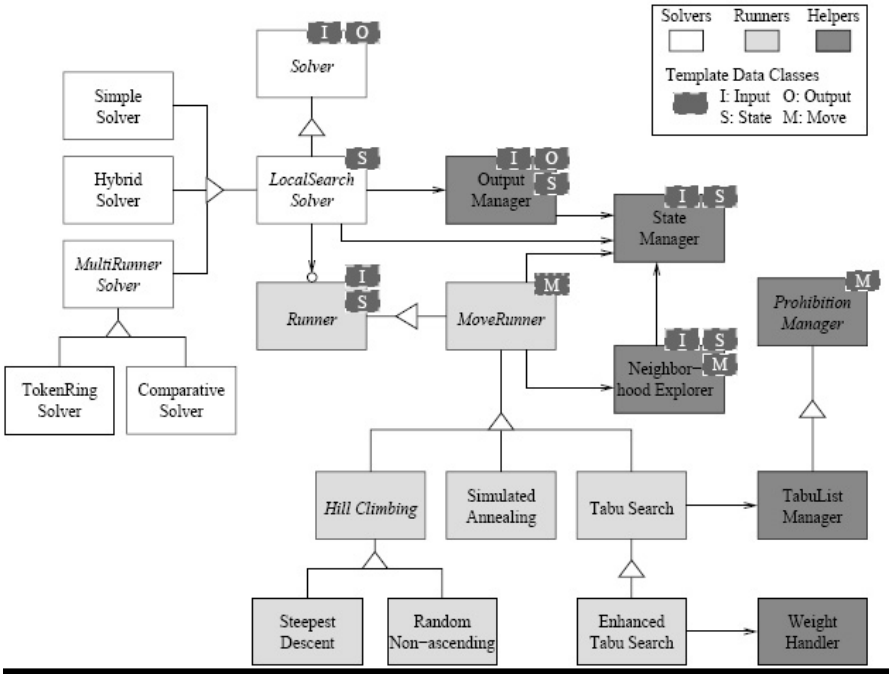


Figure 6.2: EasyLocal++main classes

solvers can be added by the user as well. This way, EasyLocal++ supports the design of new meta-heuristics and also the combination of already available algorithms.

Figure 6.1 shows an UML diagram of the main classes of EasyLocal++.

6.3 Modeling with EasyLocal++

Modeling a problem using EasyLocal++ requires that the C++ classes representing the problem specific layers of the EasyLocal++ hierarchy, i.e. Data and Helpers classes, be defined. The Data components are defined into ad-hoc classes while Helpers components are defined as subclasses of EasyLocal++ abstract classes. Once the required Data and Helper classes are implemented the user selects the desired LS algorithm from the *Runners* and *Solvers* classes and executes it. In this section we show the implementation of the main Data and Helper classes for the SCTT problem. To sum up we also give an example of a main code that uses the classes defined in order to build a LS algorithm.

6.3.1 State

The following code shows a possible implementation of a State class for the SCTT problem. Lines 5–7 declare the members of the class. We assume that all the instance information are

stored in an Input class named `Faculty`. The object `in` belongs to this class and is the member defined in line 5. Then line 6 declares a matrix `T` for the timetable that will store the values of the variables. The third member is the vector `working_days` that will store the number of working days for each course.

The two constructors are defined in lines 12–21: the first one takes as input argument an object of the `Faculty` class, while the second one takes as input another `TT_State` object. Lines 23–27 defines the copy assignment operator in order to copy a State into another one. It is worth noting that the methods defined so far are all necessary in each EasyLocal++ state class. Then lines 28–36 defines operator and methods to read and modify the members of the class.

```
(1)     #include "faculty.hh"
(2)     class TT_State
(3)     {
(4)     protected:
(5)        const Faculty & in;
(6)        vector<vector<unsigned> > T; // (courses X periods) timetable matrix
(7)        vector<unsigned> working_days; // number of days of lecture per course
(8)
(9)     public:
(10)       TT_State(const Faculty &f):
(11)          in(f),
(12)          T(in.Courses(), vector<unsigned>(in.Periods(),0)),
(13)          working_days(in.Courses())
(14)       {}
(15)       TT_State(const TT_State& s):
(16)          in(s.in),
(17)          T(s.T),
(18)          working_days(s.working_days)
(19)       {}
(20)       TT_State& operator=(const TT_State& s)
(21)       {
(22)          T = s.T;
(23)          working_days = s.working_days;
(24)       }
(25)       unsigned operator()(unsigned i, unsigned j) const { return T[i][j]; }
(26)       unsigned& operator()(unsigned i, unsigned j) { return T[i][j]; }
(27)
(28)       Faculty Input() const { return in; }
(29)
(30)       unsigned WorkingDays(unsigned i) const { return working_days[i]; }
(31)
(32)       void ResetWorkingDays(unsigned i) { working_days[i] = 0; }
(33)       void IncWorkingDays(unsigned i) { working_days[i]++; }
(34)       void DecWorkingDays(unsigned i) { working_days[i]--; }
(35)    };
```

6.3.2 Move

The following class implements the move defined in Section 3.5 for the SCTT problem, i.e., the swapping move. This move is determined by two variables of the timetable matrix that

will be swapped when the move will be applied. Thus a move is identified by two couples (*course*, *period*) that determine the two variables. Line 4–5 declares the members object of the move class, i.e., `course1` and `period1` to identify the first variable, and `course2` and `period2` to identify the second one.

The constructor methods are defined in lines 7–12, that set the values for all the members of the class. Then lines 14–34 define three operators for comparing the moves. The first one determines if two given moves are identical, i.e. if the four members have the same values among the two moves. The second method determines if two moves are different. The last method provide a comparison between two moves.

```
(1)     class TT_Swap
(2)     {
(3)     public:
(4)        unsigned course1, period1;
(5)        unsigned course2, period2;
(6)
(7)        TT_Swap( unsigned c1, unsigned p1, unsigned c2, unsigned p2 ):
(8)           course1(c1),
(9)           period1(p1),
(10)          course2(c2),
(11)          period2(p2)
(12)       {}
(13)
(14)       bool operator==(const TT_Swap& mv1, const TT_Swap& mv2)
(15)       {
(16)          return mv1.course1 == mv2.course1 && mv1.period1 == mv2.period1
(17)             && mv1.course2 == mv2.course2 && mv1.period2 == mv2.period2;
(18)       }
(19)
(20)       bool operator!=(const TT_Swap& mv1, const TT_Swap& mv2)
(21)       {
(22)          return mv1.course1 != mv2.course1 || mv1.period1 != mv2.period1
(23)             || mv1.course2 != mv2.course2 || mv1.period2 != mv2.period2;
(24)       }
(25)
(26)       bool operator<(const TT_Swap& mv1, const TT_Swap& mv2)
(27)       {
(28)          return (mv1.course1 < mv2.course1)
(29)             || (mv1.course1 == mv2.course1 && mv1.period1 < mv2.period1)
(30)             || (mv1.course1 == mv2.course1 && mv1.period1 == mv2.period1
(31)                   && mv1.course2 < mv2.course2)
(32)             || (mv1.course1 == mv2.course1 && mv1.period1 == mv2.period1
(33)                   && mv1.course2 == mv2.course2 && mv1.period2 < mv2.period2);
(34)    }
```

6.3.3 Constraints and Objective Function

Constraints in EasyLocal++ are treated as components of the objective function: they are relaxed the number of violations for each constraint is counted and contributes to increase the objective function values. The components of the objective function coming from relaxed constraints are called *hard constraints*, while the other components are called *soft constraints*,

according to the terminology given in 1.1. The contribution of hard constraints to the overall objective function values is much higher than the one of soft constraints. Thus, during the execution of any LS algorithm, solution that decrease the violations of hard constraints are preferred.

Each component (hard or soft) of the objective function is defined as a subclass of the `CostComponent` abstract class. Each component needs to define the function `ComputeCost` that, given a State object as input, calculates the contribution to the objective function from that component. Subsequently, each `CostComponent` will be linked to a `StateManager` object. When asked for the calculus of the objective function, the `StateManager` will compute the contribution of each single `CostComponent` and provide a weighted sum of them. In the code below we show the definition of the soft constraint for the SCTT problem, that refers to the number of working days involved for each course (the definition of the other components is similar). The class `MinWorkingDays` is defined, that extends `CostComponent` (line 1). The constructor is defined in lines 4–6. It passes to the superclass constructor the input object `in`, the weight for this component, a Boolean value stating if it is an hard component, a string with the name of the component.

The method `ComputeCost` is shown in lines 7–14. It takes as input argument a state and returns an integer representing the cost of this component.

```
(1)    class MinWorkingDays:  public CostComponent<Faculty,TT_State>
(2)    {
(3)    public:
(4)       MinWorkingDays(const Faculty & in, int w, bool hard):
(5)          CostComponent<Faculty,TT_State>(in,w,hard,"MinWorkingDays")
(6)          {}
(7)       int ComputeCost(const TT_State& st) const
(8)       {
(9)          unsigned c, cost = 0;
(10)         for (c = 0; c < in.Courses(); c++)
(11)            if (as.WorkingDays(c) < in.CourseVector(c).MinWorkingDays())
(12)               cost += in.CourseVector(c).MinWorkingDays() - as.WorkingDays(c);
(13)         return cost;
(14)      }
(15)   }
```

6.3.4 NeighborhoodExplorer

The NeighborExplorer Helper is responsible for the exploration of the neighborhood. Thus a class that implements the NeighborhoodExplorer functionalities needs to define the operations to select, enumerate and perform moves. The code below shows the definition a NeighborhoodExplorer for the SCTT problem corresponding to the definition of the swapping move. `RandomMove` method defined in lines 3–16 generates a random move for the state `as` passed as input. The move generated is stored in the variable referred by the input argument `mv`. The `FeasibleMove` methods (lines 18–25) takes as input a state and a move and checks if the move is not admissible, or trivial, for that state. Non feasible moves will be discarded by the LS engine during the exploration. The `MakeMove` method defined in lines 27–32 actually performs the move. It modifies the state `as`, by swapping the variables indi-

cated by the move `mv`. The last two methods are used to enumerating all the possible moves from a given State. The `FirstMove` method set the argument `mv` to the initial move. Then the `NextMove` "increases" the move in `mv` according to a specific ordering criterion defined by the user. Thus a strategy that wants to explore the whole neighborhood of a state, will first generate the starting move with `FirsMove`, then use `NextMove` that will cyclically generate all the other possible moves, until a move equal to the first one is generated.

```
(1)    class TT_SwapNeighborhoodExplorer:public NeighborhoodExplorer<Faculty,TT_State,TT_Swap>
(2)    {
(3)       void RandomMove(const TT_State& as, TT_Swap& mv) const
(4)       {
(5)          unsigned course, period1, period2;
(6)          course = Random::Int(0,in.Courses()-1);
(7)          period1 = Random::Int(0,in.Periods()-1);
(8)          do{
(9)             period2 = Random::Int(0,in.Periods()-1);
(10)         }while ( mv.period1 == mv.period2 );
(11)
(12)         mv.course1 = course;
(13)         mv.course2 = course;
(14)         mv.period1 = period1;
(15)         mv.period2 = period2;
(16)      }    bool FeasibleMove(const TT_State& as, const TT_Swap& mv) const
(17)      {
(18)         return   (as(mv.course1,mv.period1) == 0
(19)                  || in.Available(mv.course2,mv.period2) == 0)
(20)               &&
(21)                  (as(mv.course2,mv.period2) == 0
(22)                  || in.Available(mv.course1,mv.period21 == 0);
(23)      }
(24)      void MakeMove(TT_State& as, const TT_Swap& mv) const
(25)      {
(26)         int tmp = as(mv.course1,mv.period1)
(27)         as(mv.course1,mv.period1) = as(mv.course2,mv.period2)
(28)         as(mv.course2,mv.period2) = tmp;
(29)      }
(30)      void FirstMove(const TT_State& as, TT_Swap& mv) const
(31)      {
(32)         mv.course1 = 0;
(33)         mv.course2 = 0;
(34)         mv.period1 = 0;
(35)         mv.period2 = 1;
(36)      }
(37)      bool NextMove(const TT_State& as, TT_Swap& mv) const
(38)      {
(39)         if (NextPeriod2(as,mv))
(40)            return true;
(41)         else if (NextPeriod1(as,mv))
(42)            return true;
(43)         else
(44)            do
(45)            {
(46)               mv.course1++;
(47)               mv.course2++;
(48)            }
(49)            while (mv.course1 < in.Courses() - 1 && !FirstPeriod1(as,mv));
(50)         return mv.course1 < in.Courses() - 1;
(51)      }
(52)   }
```

6.3.5 StateManager

The StateManager helper is responsible for all the operations on the State that are independent from the neighborhood definition. Therefore, no Move definition is supplied to the StateManager. A class that defines the StateManager functionalities has to implement the method `RandomState`. This method generates a random solution and stores it in the state object referred by the input argument. An example of such a method is shown in lines 3–19. It is not mandatory that the State be generated in a random fashion. Heuristically generated states are also allowed. The state can be generated heuristically instead of randomly. In this class is also possible to define the method `LowerBoundReached` (lines 21–24): it takes as input an integer values representing the cost of a solution and checks if a known lower bound for the problem has been reached. In our example the lower bound is set to 15.

```
(1)    class TT_StateManager :  public StateManager<Faculty,TT_State>
(2)    {
(3)       void RandomState(TT_State& as)
(4)       {
(5)          ResetState(as);
(6)          for (unsigned c = 0; c < in.Courses(); c++)
(7)          {
(8)             unsigned lectures = in.CourseVector(c).Lectures();
(9)             for (unsigned j = 0; j < lectures; j++)
(10)            {
(11)               unsigned p;
(12)               do // cycle until the period is free and available
(13)                  p = Random::Int(0,in.Periods()-1);
(14)               while (as(c,p) != 0 || !in.Available(c,p));
(15)               as(c,p) = Random::Int(1,in.Rooms());
(16)            }
(17)         }
(18)         as.UpdateRedundantStateData();
(19)      }
(20)
(21)      bool LowerBoundReached(const int& fvalue) const
(22)      {
(23)         return fvalue <= 15;
(24)      }
(25)   }
```

6.3.6 Main

Once all the necessary Data and Helpers classes have been defined, their instance must be properly created and utilized in a main file. This main file creates the instances, links them together, sets up the search engine, starts the search and retrieves the result. An example of main file is given in the following code.

Lines 3–4 instantiate the Input and Output objects, belonging respectively to the classes `Faculty` and `Timetabling`. Line 6–9 instantiate four CostComponent: three hard constraints (lines 6–8) and one soft constraint (line 9).

Lines 11–14 instantiate four DeltaCostComponent, each for a every cost component. A DeltaCostComponent class is the “dynamic” version of the CostComponent class: it com-

putes the difference of the objective function in a given state due to a Move passed as parameters.

Line 16 creates the StateManager. All CostComponents are attached to the StateManager (lines 17–20). Then line 22 creates the NeighborhoodExplorer and all the DeltaCostComponents are attached to it (lines 23–26). Lines 28 instantiates the OutputManager, a standard Helper class that handles delivering the results to the user. Then line 30 sets up the Hill Climbing engine, implemented in the class `HillClimbing`. This class is parametric w.r.t. the input, the state and the move classes and its constructor takes as argument the input object, the state manager, the neighborhood explorer and the name of the engine. Line 31 sets the maximum number of idle iterations allowed to 1000000. The `HillClimbing` is a Runner class w.r.t. the EasyLocal++ hierarchy. A Runner can be used alone or together with other Runners, managed by a Solver. Thus, line 33 instantiates the Solver `GeneralizedLocalSearch` and line 34 attaches the `HillClimbing` runner to this Solver. Line 35 starts the search. Once the search ends, lines 38–41 show the results on the console.

```
(1)    int main(int argc, char** argv)
(2)    {
(3)       Faculty in(instance);
(4)       Timetable out(in);
(5)
(6)       EnoughLessons ELcc(in, 1, true);
(7)       SameTeacher STcc(in, 1, true);
(8)       UnavTeacher UTcc(in, 1, true);
(9)       MinWorkingDays MWcc(in, 1, false);
(10)
(11)      DeltaEnoughLessons deltaELcc(in, 1, true);
(12)      DeltaSameTeacher deltaSTcc(in, 1, truen);
(13)      DeltaUnavTeacher deltaUTcc(in, 1, true);
(14)      DeltaMinWorkingDays deltaMWcc(in, 1, false);
(15)
(16)      TT_StateManager sm(in);
(17)      sm.AddCostComponent(MWcc);
(18)      sm.AddCostComponent(ELcc);
(19)      sm.AddCostComponent(STcc);
(20)      sm.AddCostComponent(UTcc);
(21)
(22)      TT_SwapNeighborhoodExplorer snhe(in, sm);
(23)      snhe.AddDeltaCostComponent(deltaMWcc);
(24)      snhe.AddDeltaCostComponent(deltaELcc);
(25)      snhe.AddDeltaCostComponent(deltaSTcc);
(26)      snhe.AddDeltaCostComponent(deltaUTcc);
(27)
(28)      TT_OutputManager om(in);
(29)
(30)      HillClimbing<Faculty,TT_State,TT_Swap> shc(in, sm, snhe, "Swap Hill Climbing");
(31)      shc.SetMaxIdleIteration(1000000);
(32)
(33)      GeneralizedLocalSearch<Faculty,Timetable,TT_State> s(in, sm, om, "S");
(34)      s.AddRunner(shc);
(35)
(36)      s.GeneralSolve();
(37)      out = s.GetOutput();
(38)
(39)      cout << out;
(40)      cout << "Time " << chrono.TotalTime() << endl;
(41)      cout << "Cost " << s.GetBestCost() << endl;
(42)      return 0;
(43)   }
```

7

GELATO hybrid solver

Gecode and EasyLocal++ both have similar aims, allowing users to model CSPs and COPs using an imperative C++ encoding, and to solve them efficiently; moreover, they both have a modular object-oriented architecture that heps customization and new features addition. On the other hand, they address these aims exploiting different search paradigms (Constraint Programming for Gecode and Local Search for EasyLocal++). These paradigms have different strengths, i.e. (flexibility for CP and efficiency for LS) and work with distinct basic data types and search engines, leading to different architectures.

As discussed in Chapter 4, the advantages of hybridization between LS and CP are well known. Thus, it may be desirable to have a unique environment that merges the capabilities of these tools. This environment would allow the user to easily develop meta-heuristics that explore the search space, alternating different search engines on the same problem. Starting from these considerations, we developed the hybrid tool GELATO, that integrates CP and LS in a single environment, exploiting the functionalities of Gecode and EasyLocal++. We chose the name GELATO as the acronym for **G**ecode + **E**asy**L**ocal = **A** **T**ool for **O**ptimization.

From the Gecode users point of view, GELATO can be seen as a library that adds local search features and allows a Gecode programmer to easily define LS algorithms (either the classical ones or those based on LNS) starting from Gecode constraint models. From the EasyLocal++ users perspective, GELATO can be seen as an extension that allows a programmer to delegate to a CP engine tasks related to LS search, such as finding initial solutions, exploring neighborhoods, satisfying hard constraints, etc. From a global point of view, GELATO is an environment that allows a programmer to model problems and to define hybrid solving strategies using a special high-level language (see Chapter 8) and to search for solutions using Gecode and EasyLocal++ as low level solvers.

In section 7.1 an overview of GELATO features and architecture is given, then its implementation is analyzed in depth. Namely, its internal core is presented in section 7.2 and its external interface is shown in section 7.3. In section 7.4 we give some examples of basic modeling using GELATO. High-level modeling issues are indeed postponed to Chapter 8.

7.1 GELATO Overview

GELATO functionalities are divided into two main parts: an *internal core*, that merges Gecode and EasyLocal++ together, and an *external interface*, that provides an easy way for the user to interact with the system. Figure 7.1 shows the structure of GELATO.

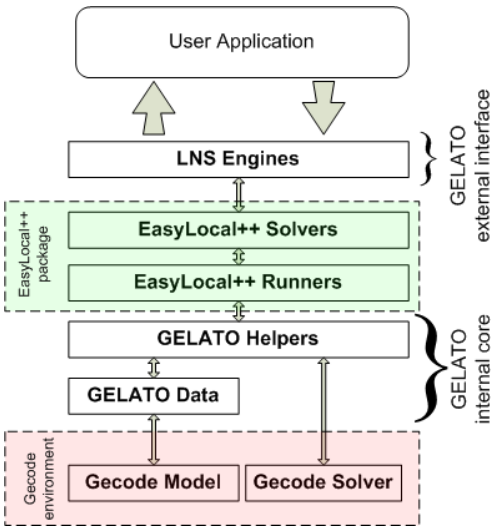


Figure 7.1: Structure of GELATO and interactions with Gecode and EasyLocal++

The *internal core* of GELATO is a *middle layer* that defines the interaction between EasyLocal++ and Gecode. Referring to the abstraction levels of EasyLocal++ (see section 6.1), it implements ad hoc EasyLocal++ classes of the *Helpers* and *Data* layers, including the encodings of Gecode models and CP search engines into these classes.

- **Data** - The classes in this layer define the basic data structure used by both EasyLocal++ and Gecode functions. They implement the four basic Data classes of EasyLocal++ (i.e., State, Move, Input, Output, see 6.2) as well as the encodings of COPs and CSPs models, implemented in Gecode or in other modeling languages (supported by the GELATO tool).

- **Helpers** - At this level the definitions of the helpers required by EasyLocal++ are stored, such as the `NieghborhoodExplorer` and `StateManager` (see 6.2). This level implements the LNS algorithms that exploit a CP solver in the exploration of the large neighborhoods; traditional LS algorithm are also provided by the standard EasyLocal++ layers.

The *external interface* of GELATO provides high level functions that can easily be called by the user to access the inner GELATO functionalities, thus hiding all the internal Gecode and EasyLocal++ calls. This external layer act as a *Façade* object-oriented pattern

(see [32]): the aim of the Facade pattern is to provide a simplified interface to a larger body of code, making the subsystem easier to use. Thus, this external layer transforms the overall GELATO tool into a black-box and allows the user to interact with it just by passing the input (i.e., the problem model and the solving meta-heuristic) and retrieving the output (i.e., the solution obtained), without caring about complicated object interactions in the GELATO internal core. In this scenario, the user is asked to:

1. **model the CSP or COP** with a modeling language supported by GELATO (in the current release, models written in Gecode as well as in the FlatZinc language are accepted);

2. **specify the instance** of the problem in a different file, since GELATO requires the concepts of *problem* and *instance* to be separated;

3. **select the meta-heuristic** to be used to find the solution (e.g., Hill Climbing,Tabu Search, Large Neighborhood Search) and specify the desired parameters for the meta-heuristic execution.

It is important to note that GELATO does not require *any modification of* Gecode *and* EasyLocal++. In this way GELATO does not affect the single development of Gecode or EasyLocal++ and every improvement of the two basic tools, such as new functionalities or performance improvements, is immediately inherited by GELATO.

In the next sections we focus with more details on the internal core and on the external interface of GELATO, providing diagrams and explanations of class interactions.

7.2 GELATO Internal Core

The internal core of GELATO mirrors the architecture of EasyLocal++. It provides the definitions of the concrete classes needed by EasyLocal++ to work, containing the specific data structures and algorithms for LNS. All these concrete classes belong to the Data and Helper layers. In the following sections we first describe the Data classes, that provide the basic data structures for GELATO and then the Helpers classes, that define algorithms operating on the Data classes. The algorithms defined in this layer implement the search strategies that hybridize the LS and CP paradigm, exploiting the Gecode constraint environment.

7.2.1 Data

At this level the basic data structures that are used by the GELATO tool are defined. Since EasyLocal++ needs an implementation of four basic classes, i.e., State, Move, Input and Output, ad hoc implementations of these classes are provided. Moreover, a class that wraps the abstract functionalities of a Constraint Programming model (such as, variables, constraints, objective functions) is defined. This class is called `GelatoModel` and its concrete implementation may refer to different modeling languages, e.g., Gecode or FlatZinc. Let us now analyze in detail each single class.

GelatoState

This data structure represents a single *solution* (or *state*, according to the EasyLocal++ terminology) of the problem $\mathcal{P}$ we are solving (as defined in section 1.1). A GelatoState object can be accessed (read or modified) by both the Gecode environment and the EasyLocal++ framework. Since it represents a solution, an instance of this class contains the variables of the problem with the associated values. The variables are organized in several vectors and every vector is loaded in a map (instance of the class `VarMap`), with an associated key. One of the arrays is expected to contain the variables that the search module will branch on, so the name of such a vector is stored in the `branchingArrayName` string object.

In the current implementation, the variables vectors are IntegerVarArray objects (standard Gecode data structures for arrays of integer variables) and the keys are strings of characters. A typical GelatoState object contains just a single array of integer variables accessed by the key `"x"`, which also represents the array that will be branched on.

A GelatoState object also contains the Input object of the current instance being solved and the cost of the state according to objective function definition. Even if state and FObj are separate concepts and EasyLocal++ originally does not store any cost information into the state, we choose to memorize the cost of a state into the state itself for efficiency reasons: computing the objective function of a state is an expensive operation, so once the cost is computed and stored in the state, it can be quickly retrieved without any recomputation if needed in the future.

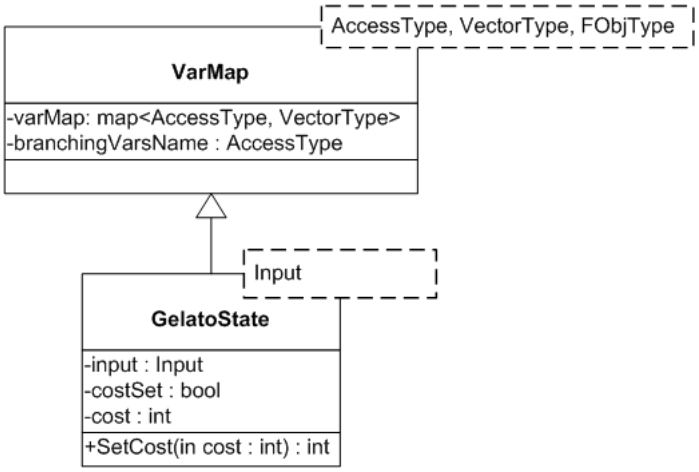


Figure 7.2: GelatoState class

Move

The abstract class Move represents a LS move used by the GELATO hybrid search engine, thereby representing a Large Neighborhood Search move. A LNS move defines the sets FV of *free variables*, according to the definition given in section 4.2. The basic implementation

of a *FV* set is given by the class `NFreeVars`, which collects the names of the free variables into a vector. In this way, checking if a variable is free w.r.t. a certain move is performed by verifying if the name of the variable is contained into the `NFreeVars` vector.

A `GelatoMove` is a more complex object that refers to one or more instances of NFreeVars and stores the following information:

1. the variables involved in the move;
2. if the move has been already tried or not. If the move has been tried, the following information are stored:
   (a) the state reached by the move, i.e. the new value of the variables after the execution of the move;
   (b) the FObj value of the state reached by the move.

Information 1) can be retrieved using the operation `containsVar( varName )`, that takes as input the name of a variable and may return true, if the variable is involved in the move, or false, otherwise. Information 2) is stored in the boolean variable `varsAssigned` and information 2a) and 2b) can be obtained respectively by reading the values of variable `reachedStateCost` and vector `assignmentVars`. Every concrete `GelatoMove` class must also provide methods for comparing two move instances, i.e. checking if two moves are equal or different (operation performed by `operator==()` and `operator!=()`) and giving an ordering of the moves (operation performed by `operator<()`).

Note that in a `GelatoMove` class no information about *how* to perform the move is stored, but only information about the characteristics of the move.

We have described the characteristics of an abstract `GelatoMove`, i.e., information and operations that every move must provide. On the basis of this common characteristics it is possible to define several kinds of concrete `GelatoMove`. Two of them has been developed for programmer's convenience:

- `NDifferentMove`: it is defined by a set *FV* of free variables (i.e., an instance of `NFreeVars`) and a positive integer number *n*. In this move, only *n* of the *FV* can be modified by a LS search engine. The *n* names of the *n* modifiable variables are stored in the vector `varsSelection`. This kind of move deals with a subset of *FV* having fixed cardinality but not fixed elements.

- `DistrictMove`: it is defined by several sets of variables (i.e., several `NFreeVars` instances, collected into a vector), that represents different neighborhoods. At any given time only the variables belonging to a specific set are free, while the other ones are fixed. The set (i.e., the neighborhood) containing the free variables is called *active*. Thus, we have one active set at time, that contains the variables that can be modified by the LS move execution. Each set can become active at some point, allowing a cyclic exploration of the neighborhoods specified by the sets.

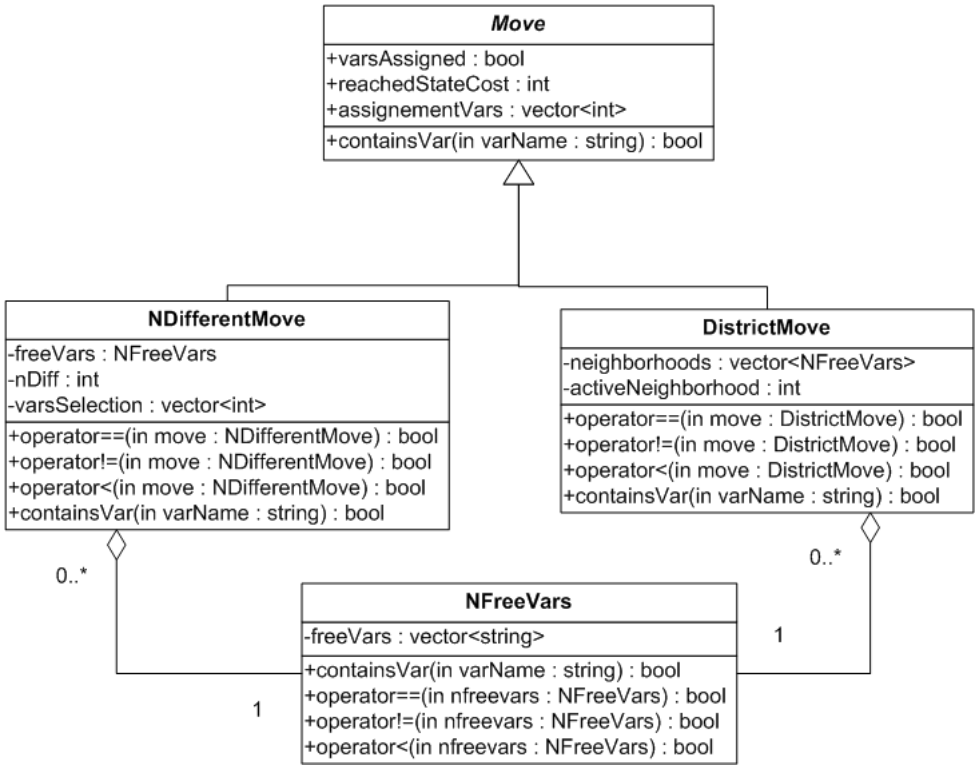


Figure 7.3: Gelato Move class hierarchy

GelatoInput

In the GELATO framework, the concepts of *problem*, and *instance* are separated, as requested by the EasyLocal++ framework. The specific instance of a problem is defined by a subclass of the abstract class `GEInput`. The abstract class provides basic information, such as the size of the problem and the dimension of the variables array, each respectively stored in the Gecode data structure `SizeOptions` and in the integer variable `dimension`. There are two main types of concrete Input classes:

- **Gecode input** classes: these classes extend the abstract class `GEInput` and define ad hoc information for a specific Gecode model. For instance, the `TSPInput` class that models the instances of the Travel Salesman Problem typically contains the distance matrix. The `CTTInput` for the Course Time Tabling problem may contain information about number of teachers, courses, rooms, availabilities, and so on.

- FlatZinc **input** class: this class provides input for models written in the FlatZinc format (see section 8.2). FlatZinc mixes definition of problem and instance together, so this class contains the reference to a `.fzn` file where the information of a problem and

an instance are collected.

It is worth to comment that every concrete `GelatoInput` class has to be recognizable from the respective Constraint Programming problem definition, stored in a `GelatoSpace` class (whose hierarchy is explained hereinafter in this section). For instance, the class `GEMinModel` will construct a correct Gecode minimization model from both the general problem model (contained in the `GEMinModel` class) and the specific instance (contained in the `GelatoInput` class); the class `FZModel` will construct the constraint programming model from the `.fzn` file specified in the `FZInput` class.

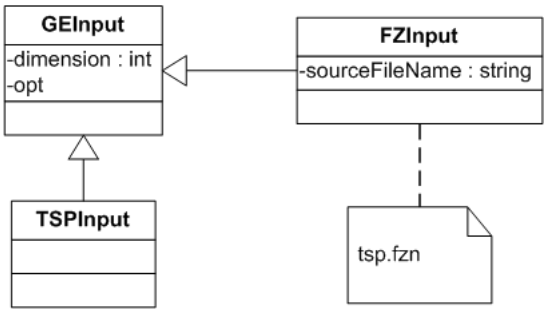


Figure 7.4: GelatoInput class

GelatoOutput

This class implements the definition of a standard Output class, as requested by EasyLocal++ in order to print the result of an execution of a Local Search algorithm. It contains a `GelatoState` instance that represents the final state at the end of the search. It also implements the `operator<<()` that prints the characteristics of the final state reached on the console.

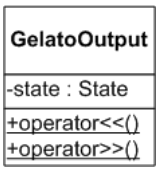


Figure 7.5: GelatoOutput class

GelatoSpace

A GelatoSpace class represents a Constraint Programming model for a particular problem. It is a repository for all the model information, including variables, domains, constraints and the objective function, as well as basic solving information, such as the branching strategy.

Information about the characteristics of a specific instance to solve is contained within an `Input` object, that will be subsequently attached to the `GelatoSpace` object at run time.

`GelatoSpace` is an abstract interface class, parametric w.r.t. the following classes: `Model` (the concrete constraint programming model, e.g., TSP), `Input` (the concrete `GelatoInput` class, e.g., `TSPInput`), `VarType` (the data type for a single variable, e.g., `IntVar` since we are using Gecode), `ArrayVarType` (the data type for an array of variables, e.g., `IntVarArray` since we are using Gecode), `Options` (the class containing parameters for the search process, e.g., `Search::Options` since we are using Gecode).

This means that in order to define a GelatoSpace concrete subclass, we have to specify a concrete class for each parametric class. Moreover, the concrete class has to implement the method requested by the GelatoSpace interface: a Local Search engine will call these interface methods during its execution, without knowing which concrete class they belong to. The methods that every concrete class of the `GelatoSpace` hierarchy must implements:

- `Model CreateNewModel( Input )`: this method creates and returns a new model from scratch, using the information contained in the `Input` parameter;
- `VarArrayType GetIntegerVars()`: this method returns the entire branching vector of variables;
- `VarType GetIntegerVar( index )`: this method returns the variable of the branching vector corresponding to the position specified by the index parameter;
- `VarType GetFObj()`: returns the variable that contains the FObj value of the model;
- `int InfinityObj()`: returns the Infinite value w.r.t. the FObj definition, e.g., a large positive integer number for a minimization FObj or a large negative integer number for a maximization FObj;
- `void postBranching( firstSolution )`: contains the statements that determine how to perform branching on the variable of the problem by taking as input a Boolean value that specifies if the branching is on the first solution or not. With this information it is possible to define different branching strategies for searching the first solution or exploring the neighborhoods.
- `Model solve( firstSolution, Options )`: this method executes the exploration of the search space defined by the `GelatoSpaceModel`. It has two parameters, i.e., `firstSolution`, and `Options`. The first one specifies if the exploration is on the first solution or not (allowing to define different strategies to find the first solution or to explore the neighborhood), while the second contains parameters for the exploration process.

In the current implementation of GELATO, we have defined three GelatoSpace classes. All these classes represent models that can be read and solved by a Gecode search engine, with the main difference being that the first and the second classes are Gecode model, while the third one is a FlatZinc model. Here follows a brief description of the three classes:

- `GEMinModel`: this class inherits from the `MinimizeSpace` class of Gecode, which is the parent class of all the maximization models in Gecode. In this way, `GEMinModel` inherits all the characteristics of a Gecode minimization model. It also defines all the functionalities requested to work on GELATO since it implements the `GelatoSpace` interface. A real minimization model for GELATO, such as a model for the TSP problem, will be defined in a specific class that extends the `GEMinModel` class.

- `GEMaxModel`: this class mirrors the previous one, with the only difference that it inherits from the `MaximizeSpace` class of Gecode and is used to define maximization models.

- `FZModel`: this class inherits from the `FlatZincSpace` class of Gecode, which is the class of all the FlatZinc models in Gecode. `FZModel` inherits all the characteristics of a FlatZinc model in Gecode environment, and defines all the functionalities requested to work on GELATO as long as it implements the `GelatoSpace` interface. This class contains the reference to a `.fzn` file, that contains the FlatZinc model desired. Unlike the previous class, the programmer does not need to define a new subclass of `FZModel` for defining a FlatZinc model for a specific problem; he/she needs to have a `.fzn` file containing the model in FlatZinc language and to create an instance of `FZModel` specifying the name of that file.

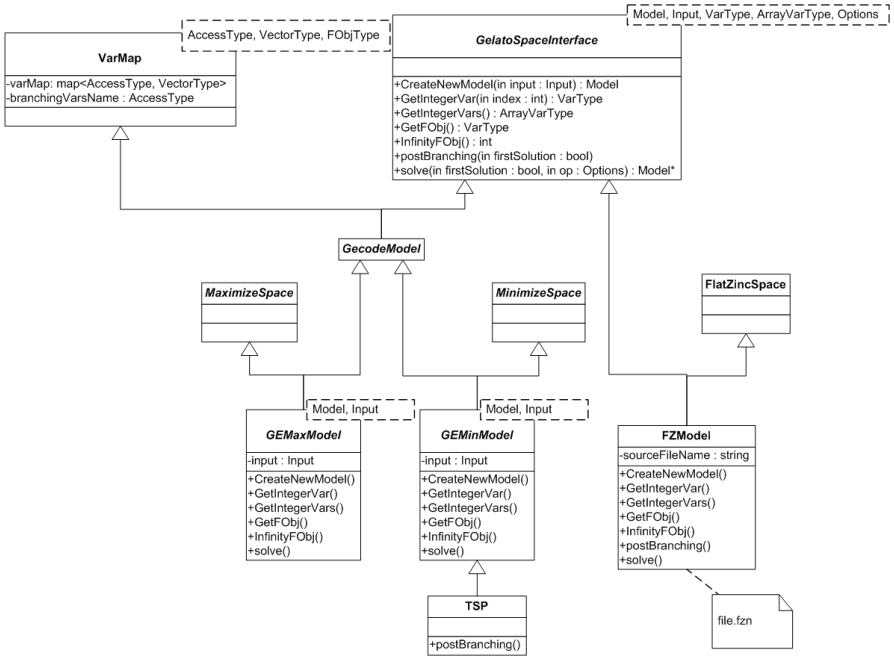


Figure 7.6: GelatoSpace class

7.2.2 Helpers

At this level the Helpers class that implements the functionalities of a LS algorithms are defined. They inherit from the EasyLocal++ abstract helper classes and provide the implementations of the requested methods.

GelatoNeighborhoodExplorer

The NeighborhoodExplorer helper is the most crucial one in the EasyLocal++ architecture, because it handles all the features concerning the neighborhood exploration, in particular how to *select* and how to *perform* a move. This class is parametric w.r.t. the Input, the State and the Move class, i.e., each LS algorithm will operate on its specific Input, State and Move class. The methods that any NeighborhoodExplorer must implement are:

1. `RandomMove(State,Move)`: generates a random move for the state `State` and stores it in the variable `Move`;
2. `FirstMove(State,Move)`: generates the first move for the state `State`, according to a neighborhood exploration strategy, and stores it in `Move`;
3. `NextMove(State,Move)`: modifies `Move` to become the candidate move that follows the current value of `Move` according to the neighborhood exploration strategy. This function, together with the previous one, is used in algorithms relying on exhaustive neighborhood exploration.
4. `MakeMove(State,Move)`: updates the state `State` by applying the move `Move` to it.

As long as GELATO deals with LNS algorithm, it has a `GelatoNeighborhoodExplorer` helper (a subclass of `NeighborhoodExplorer`), that defines the LNS neighborhoods. As explained in section 4.2 there are three aspects that characterize a specific LNS approach: *1) which variables* are free, *2) how many variables* are free (i.e., the definition of FV), and *3) how to explore* these variables.

Our `GelatoNeighborhoodExplorer` class splits the definition of these aspects into two auxiliary classes: `MoveEnumerator` and `MovePerformer`. The `MoveEnumerator` class defines functions 1, 2, and 3 of `NeighborhoodExplorer` since they all concern the *definition* FV. The `MovePerformer` class defines functionality 4, which is the only one that determines how to *explore* the free variables. Let us analyze these two classes in more details.

The `MoveEnumerator` (or *ME* for short) deals with the definition of the set $FV \subset \mathcal{X}$ for the LNS algorithm, specifying which variables of the constraint model will be free in the exploration of the neighborhood. There are many possible strategies for enumerating moves, that lead to many concrete subclasses of `MoveEnumerator`. For example we can specify a *random ME* that randomly selects a specified number of problem variables, an *iterative ME* that iterates among all the combinations of the problem variables of a given fixed size, or a *sliding ME* that considers a sliding window of k variables (i.e., $FV_1 = \{X_1, \ldots, X_k\}$, $FV_2 = \{X_2, \ldots, X_{k+1}\}$, $\ldots$). A particular ME usually works on a specific

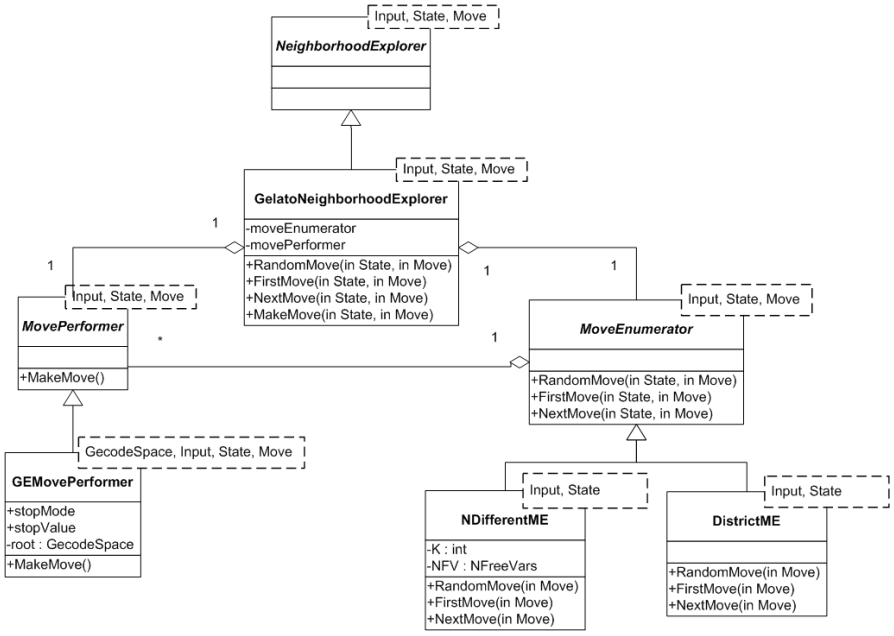


Figure 7.7: GelatoNeighborhoodExplorer class

concrete move. In the current version of GELATO, two types of moves are provided, i.e., `NDifferentMove` and `DistrictMove` as well as their respective MoveEnumerators: `NDifferentMoveEnumerator` and `DistrictMoveEnumerator`.

The `MovePerformer` (or *MP* for short) is a class that collects all the information about how to search a large neighborhood, such as variable and value selection, constraint propagation policy, branching strategies, search timeout and so on. Moreover, the MP actually executes the move, calling a search engine on a concrete instance of a problem model. The MP invokes the CP solver passing all the necessary information and obtain the result of the exploration. In the current implementation of GELATO we have defined just one concrete MP. Our concrete implementation exploits the Gecode environment, as it instantiates a `GelatoSpace` model and runs a Gecode exploration on it.

GelatoStateManager

The StateManager helper is responsible for all the *operations on the state* that are independent of the neighborhood definition; therefore, StateManager does not have any interaction with the Move data structures. GELATO defines its own implementation of the StateManger helper in the class `GelatoStateManager`, which is parametric w.r.t. the GelatoSpace, the Input, and the State classes. `GelatoStateManager` contains information about the constraint programming model to solve (stored in the GelatoSpace class), the input of the specific instance of the problem (stored in an instance of the Input class), the State data type

(stored in the State class) and a possible lower bound for the problem (stored into an integer member variable). `GelatoStateManager` implements the following methods:

- `RandomState( State )`: finds an initial state and stores it in the variable `State`. This method call the Gecode search engines that runs a naive search process on the space defined by the GelatoSpace model and the respective input. The state reached in this exploration will be used as a starting point for the subsequent LNS exploration.
- `LowerBoundReached( Value )`: returns true if the objective function value `Value` has reached the lower bound for the problem, false otherwise. It is called by the LNS process, that can stop the search if the lower bound is reached. This method actually works only if a lower bound information is set in the `GelatoStateManager`.

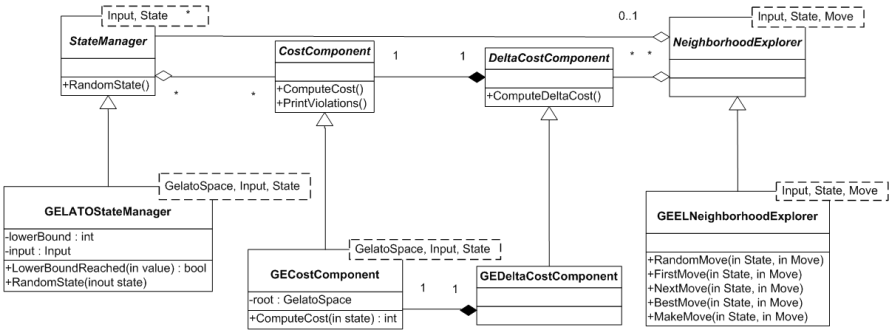


Figure 7.8: Main Gelato Helpers classes

GelatoCostComponent

The CostComponent helper handles the definition of the objective function. In detail, it is owned by the StateManager and computes the objective function on a given state, assessing the quality of the state. GELATO defines a CostComponent helper named `GECostComponent`, which is parametric w.r.t. the GelatoSpace, the Input, and the State class. `GECostComponent` implements the following method:

- `int ComputeCost( State )`: returns the objective function value of the state `State`.

This computation relies on the objective function definition stored in the GelatoSpace model. As previously explained, every GelatoSpace model has an instance variable that contains the $fObj$ value, that can be retrieved via the `GetFobj()` method. Thus, the simplest way to calculate the objective function value for a given state is: 1) to instantiate a new Gecode model, 2) to set the variable values according to the values contained in the state, 3) to run the propagation process on the model 4) to read the objective function value on the designate variable. Step 1) is the same for all the execution of `ComputeCost( State )`,

since the Gecode model remains the same, and the only thing that changes is the parameter state. Thus, instead of creating a new Gecode model every time form scratch (which is an expensive procedure), a Gecode model is created once for all and stored in the variable `root` of the `GECostComponent`. Every time `ComputeCost( State )` is called, steps 2) to 4) are applied on the model contained in `root`, obtaining a more efficient procedure.

EasyLocal++ also request the definition of a `DeltaCostComponet` helper: it is the dynamic companion of the `CostComponet` helper. It computes the difference of the cost function in a given state due to a Move passed as parameter and it is attached to a suitable Neighborhood Explorer from which it is invoked. This helper comes in EasyLocal++ with a default implementation, that has not been changed in the GELATO framework. Thus, GELATO just define its own `GEDeltaCostComponet`, wich recall the basic EasyLocal++ definition.

GelatoOutputManager

The OutputManager helper is responsible for translating between elements of the search space and output solutions. It also delivers other output information about the search, and stores and retrieves solutions from files. It is parametric w.r.t. the Input, Output and State classes, and this is the only helper that deals with the Output class. Since OutputManager comes with default definition in EasyLocal++, the GELATO implementation of this helper (named `GelatoOutputManager`), just to recall the basic functionalities. The main method of `GelatoOutputManager` are:

- `InputState( State, Output )`: gets the state `State` from the output `Output`.
- `OutputState( State, Output )`: writes the output object `Output` from the state `State`.

7.3 GELATO External Interface

Once the Data and the Helper layers are defined, EasyLocal++ provides all the control structures for Local Search algorithms in the Solver and Runner layers, which are not redefined by GELATO. The Solver and Runner layers define the high level interactions between the above components, building up a real LNS algorithm. This finds an initial solution using the `GelatoStateManager`, and then navigates the search space through the neighborhoods defined by the Move Enumerator. The neighborhoods are explored using the Move Performer. Any LS strategy can be used at this step (e.g., Hill Climbing, Steepest Descent, Tabu Search). Moreover, the combination of different LS algorithm is allowed (e.g., Multi-Neighborhood, Token Ring Search) when different kinds of ME and MP are defined for the same problem. Modularity and extendibility of EasyLocal++ are directly inherited by GELATO.

A programmer with a good knowledge of EasyLocal++ can now build up his own LNS algorithms, instantiating the desired solver with the correct objects of the GELATO internal core using the correct EasyLocal++ statements. In order to simplify the work for non-expert users of EasyLocal++, an external interface sets up the solver and hides all the

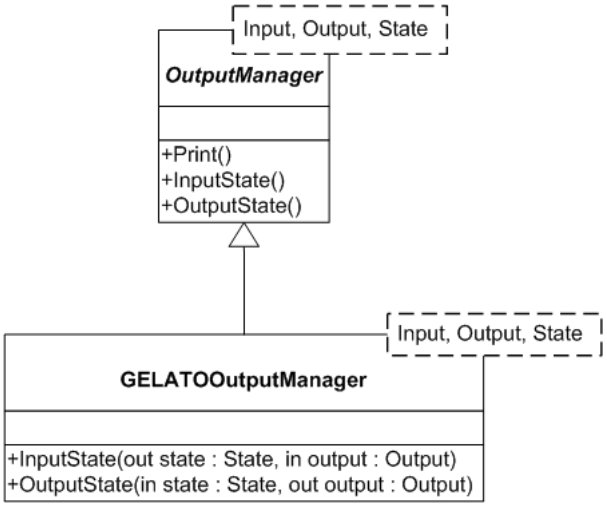


Figure 7.9: GelatoOutputManager class

interaction between the internal components of GELATO. This external layer provides the programmer a default way to call the GELATO functionalities at high-level, without caring about inner objects declaration and calls. The external interface interacts with the user only in terms of input and output. The *input* requested is the instance to solve and the search parameters. The *output* is return is made of search information written on the console and the final solution written in a file.

The external interface provides a class for each possible Large Neighborhood Search engine that exploits the implementation of the internal core of GELATO. In the current version of GELATO a Hill Climbing LNS engine is provided. The hierarchy of the LNS engines as well as the Hill Climbing engine are explained in the next section. Thus a brief description about how to call the engine from the main program is given.

7.3.1 GELATO Large Neighborhood Search Engines

All the LNS engines inherit the basic structure from the parent abstract class `LNSEngines` as figure 7.10 shows. This class is parametric w.r.t. the `GelatoSpace` and the `Input` classes that define respectively the model and the instance of the problem to be solved. The `LNSEngines` class contains the input instances, stored in the variable `input`, as well as the name of the output file, stored in the string `outputfilename`, where the solution will be written at the end of the search.

The abstract class contains the virtual method `Go( observe )` that each concrete subclass is asked to define. This method specifies which operations will be run in the GELATO internal core in order to execute the LNS algorithm and solve the given problem. The Boolean parameter `observe` specifies if the user wants to constantly observe the solver during its execution: if `true`, every time a better solution is found, a message is printed on the console,

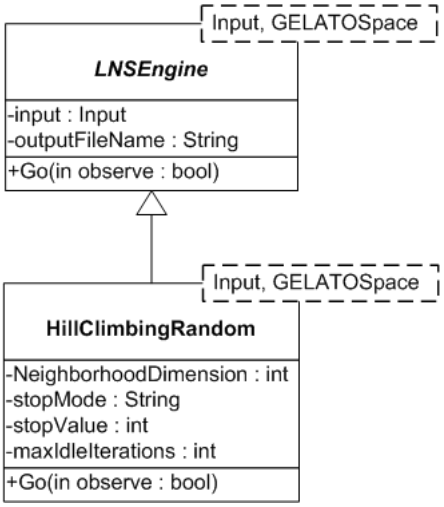


Figure 7.10: LNS Engines classes

containing the value of the new solution and the time passed from the starting of the search.

HillClimbingRandom In the current version of GELATO, only a Hill Climbing engine for LNS is provided, called `HillClimbingRandom`. This engine models a LNS algorithm that at each step randomly selects the set of free variables FV and explores the relative neighborhood using a Gecode branch and bound search. The algorithm searches for a better solution and once this is found, the current solution is updated. Each neighborhood is explored until a better solution is found or a stop criterion is satisfied. Then a new random neighborhood is selected and explored. The whole process stops when a certain number of idle iterations has been performed.

The class that models this algorithm contains the following additional search parameters:

- `NeighborhoodDimension`: is an integer variable that represents the cardinality of the set FV of free variables. It is fixed during the execution of the search.
- `stopMode`: is a string that can have either the value `TIME` or `FAIL`. In the first case each neighborhood is explored for a certain amount of time; in the second case each neighborhood is explored until a certain number of failures (inconsistent states that force backtrack).
- `stopValue`: if `stopMode` is set as `TIME`, this integer variable contains the number of milliseconds allowed for the neighborhood exploration. If `stopMode` is set as `FAIL` this integer variable contains the number of failures allowed.
- `maxIdleIterations`: is an integer variable that represent the stop criterion for the whole LNS algorithm. The search will stop after `maxIdleIterations` idle iterations has been performed.

In order to execute the search, the `HillClimbingRandom` class define its implementation of the `Go( observe )` method. This method creates an instance for each basic component (i.e., `GECostComponent`, `GEStateManager`, `GELATOState`, `GEMovePerformer`, `NDifferentME`, `GELATONeighborhoodExplorer`, `GELATOOutputManager`) and declares a Hill Climbing solver that will use all of these components. Then the method runs the search and writes the result on the specified output file.

7.3.2 Calling the external interface from the `main` program

In this section we show how to use the external interface in a C++ code. Figure 7.11 shows the code of a main program calling GELATO to solve a TSP instance. We assume that the user runs the program (named `runmain.exe`) passing the following six parameters at the console:

1. `Instance`: the number of the TSP instance to be solved; data for six instances are already loaded in the `TSPInput` class, thus this parameter can have value from 0 to 5;
2. `NeighborhoodDimension`: value for `HillClimbingRandom` engine parameter;
3. `stopMode`: value for `HillClimbingRandom` engine parameter;
4. `StopValue`: value for `HillClimbingRandom` engine parameter;
5. `maxIdleIterations`: value for `HillClimbingRandom` engine parameter;
6. `outputfilename`: name of the output file, for the `HillClimbingRandom` engine.

So a possible command to run the program is:

```
$ runmain.exe 3 30 TIME 10000 100 output.txt
```

Now let us analyze the code of the `main` program. First a SizeOptions `opt` object is declared and setup (lines 3–6). SizeOptions is a standard Gecode class that contains some options for the CP environment. Then, on line 8, the input object is instantiated. All the command line parameters are read and put into appropriate variables (lines 10–14). On lines 16–23 an instance of the `HillClimbingRandom` engine is created, passing the input and output information, as well as the search parameters. Note that in the instantiation of the `HillClimbingRandom` object, the classes representing the problem model and the input are passed as parameters (using the angle brackets). At the end (line 24), the engine is executed, via the method `Go`, specifying to observe the search process.

7.4 Modeling with GELATO

As explained in the previous section, GELATO is able to solve models encoded either in Gecode or in FlatZinc. Usually, a FlatZinc model is not directly defined by a programmer,

```
(1)    int main(int argc, char* argv[])
(2)    {
(3)       SizeOptions opt("TSP");
(4)       opt.solutions(0);
(5)       opt.icl(ICL_DOM);
(6)       opt.parse(argc,argv);
(7)
(8)       GEInput* input = new TSPInput( &opt, atoi( argv[1] ) );
(9)
(10)      int NeighborhoodDimension = atoi( argv[2] );
(11)      char* stopMode = argv[3];
(12)      int stopValue = atoi( argv[4] );
(13)      int maxIdleIteration = atoi( argv[5] );
(14)      char* outputFileName = argv[6];
(15)
(16)      HillClimbingRandom< GEMinModel<TSP,GEInput>,GEInput> hcg(
(17)         *input,
(18)         NeighborhoodDimension,
(19)         stopMode,
(20)         stopValue,
(21)         maxIdleIteration,
(22)         outputFileName
(23)      );
(24)      hcg.Go( true );
(25)   }
```

Figure 7.11: `main` code that calls the `HillClimbingRandom` engine

but it is the results of a compilation process from a high level language (this topic will be deeply discussed in the next chapter). Thus, in this section we focus on Gecode models. We first show how to define a Gecode model for GELATO and then a main code that exploits this model and set up an hybrid GELATO search engine.

7.4.1 A Gecode model for GELATO

Section 5.3 shows a Gecode model for the SCTT problem. In this section we will show how to adapt that model to make it work with GELATO. Only two slight modifications are needed w.r.t. the original Gecode encoding, thus requiring a minimum programming effort. These modifications can be summarized in:

- binding the class of the model to the GELATO tool;
- moving the branching strategy outside the model definition.

Class Binding Standard Gecode models typically inherit from `MinimizeSpace` or `MaximizeSpace`. In order to work with GELATO, models must inherit from `GEMinModel` or `GEMaxModel`. However, according to the hierarchy shown in Figure 7.6, the model class indirectly inherits from `MinimizeSpace` or `MaximizeSpace`. Moreover the array containing the branching variable of the Gecode model must be specified. In this way the GELATO solver will be able to construct a proper `GelatoState` object, bind it to the model and execute the search on the appropriate variables.

Thus the model head shown in Section 5.3.1 needs to be modified in this way:

```
(1)     class Timetabling :  GEMinModel<Timetabling,Faculty> {
(2)        IntVarArray x;
(3)        IntVar fobj;
(4)        Timetabling(const Faculty& in ):
(5)           x( *this, in.Courses() * in.Periods() , 0, 1 ),
(6)           fobj( *this, 0, Int::Limits::max )
(7)        {
(8)           branchingVarsName="x";
(9)           varMap["x"] = x;
(10)                    ...
```

Line 1 sets the proper superclass. Note that the `GEMinModel` is parametric w.r.t. the model class (in this case `Timetabling`) and the input class (in this case `Faculty`). Lines 8–9 set the name and the array of the branching variables. Also the head of other auxiliary constructors has to be modified in this way.

Branching strategy Standard Gecode models usually include statements for the branching strategy inside the model definitions. GELATO needs the model and the branching strategy to be separated. This leads to several advantages, such as executing model propagation and model branching in different moments, specifying different strategies for different searches, and so on.

Thus the statement specifying the branching strategy in the model must be deleted. In our case we delete the statement `branch` in line 60 of the code in Section 5.3.3. The branching strategy will be specified in the `postBranchings` method, defined inside the class of the model as follows:

```
(1)     void postBranchings(bool firstSolution)
(2)     {
(3)        if ( firstSolution )
(4)           branch(*this, succ, INT_VAR_MIN_MIN, INT_VAL_MIN);
(5)        else
(6)           branch(*this, succ, INT_VAR_DEGREE_MIN, INT_VAL_RND);
(7)     }
```

This method will be called by the GELATO search engine when the engine will need to set the branching strategy for the exploration. The statement in line 4 is executed when the search for the first solution is started, the statement in line 6 is executed in all the other cases. In this way, the programmer can specify different search strategy for the search of an initial

solution and for the exploration of the neighborhoods.

7.4.2 The main

The easiest way to set up a GELATO search engine and run the search is to use the GELATO external interface as explained in Section 7.3. However, a programmer with a good knowledge of the EasyLocal++ framework can directly access the functionalities of the GELATO internal core, creating and binding the objects for the Large Neighborhood Search in GELATO. In the following code we show an example of such code. Its structure is similar to the EasyLocal++ main file showed in Section 6.3.6. Here the Data and Helper classes used are the one provided by the GELATO library.

```
(1)    int main(int argc, char* argv[])
(2)    {
(3)    Faculty input( argv[1] );
(4)
(5)    GECostComponent<SCTTModel, Faculty, GELATOState<Faculty> > gcc(input);
(6)    GEDeltaCostComponent<Faculty,GELATOState<Faculty>,NDifferentMove> gDeltacc(input,gcc);
(7)
(8)    GEStateManager<SCTTModel, Faculty, GELATOState<Faculty> > sm( input );
(9)    sm.AddCostComponent( gcc );
(10)
(11)   GEMovePerformer<SCTTModel,Faculty,GELATOState<Faculty>,NDifferentMove>
(12)           mp( input, TIME, 50000 );
(13)
(14)   vector<string> vn = ["x(1)","x(2)",...,"x(30)"];
(15)   NFreeVars nfv( vn );
(16)   NDifferentME<Faculty,GELATOState<Faculty>>
(17)           nDiffME( 10 , nfv, &mp );
(18)
(19)   GelatoNeighborhoodExplorer<Faculty,GELATOState<Faculty>,NDifferentMove>
(20)           ne( input, sm, "NE STRING", nDiffME, mp);
(21)   ne.AddDeltaCostComponent( gDeltacc );
(22)
(23)   GEELOutputManager<Faculty,GELATOOutput<GELATOState<Faculty>>,GELATOState<Input>>
(24)           om( input );
(25)
(26)   HillClimbing<Faculty,GELATOState<Faculty>,NDifferentMove>
(27)           hc( input, sm, ne, "HC EL" );
(28)   hc.SetMaxIdleIteration( 1000000 );
(29)
(30)   GELATOState<Faculty> state( input );
(31)   sm.RandomState( state );
(32)
(33)   hc.SetState( state );
(34)   hc.Go();
(35)
(36)   int finalCost = hc.GetStateCost();
(37)   fstream file( "solution.txt", fstream::out );
(38)   file   << "\n\nSolution found: "<< hc.GetState()
(39)           << "Cost: " << hc.GetStateCost() << "\n";
(40)   file.close();
(41)   } /*end main*/
```

Line 3 creates the Input object, instance of the class `Faculty`. Lines 5–6 set up the CostComponent and the DeltaCostComponent of the GELATO solver. Line 8 instantiates the StateManager, and line 9 links the CostComponent to it. Lines 11–12 create the MovePerformer, specifying the stop criterion for the exploration of a single neighborhood, i.e. a timeout 5000 milliseconds. Lines 14–15 create an instance of the `NFreeVars` object, that is used in lines 16–17 to instantiate the MoveEnumerator. The first input argument of the MoveEnumerator specifies the cardinality of the set FV of free variables for the LNS algorithm (10 in this case). Lines 19–20 create the NeighborhoodExplorer, passing the Input, the StateManager, the MoveEnumerator and the MovePerformer objects as arguments. Then the DeltaCostComponent created in line 6 is linked to the NeighborhoodExplorer (line 21). Line 23 creates the OutputManager. Lines 26–27 instantiates the `HillClimbing` Runner of EasyLocal++ framework, using the StateManager and the NeighborhoodExplorer as input arguments. Line 28 sets the number of idle iterations allowed. Line 30 creates a State object and line 31 makes this object became a random state. Line 33 sets the random state generated as the initial state for the Hill Climbing algorithm and line 34 starts the Hill Climbing search. Lines 36–39 retrieve the solution obtained and write it on a file.

8

GELATO Meta-Tool

One of the main goals of GELATO is to make the Large Neighborhood Search paradigm easy to use for a wide range of programmers. In order to use the GELATO solver as described in section 7.4, a basic knowledge of the C++ programming language is required, as well as a good knowledge of the Gecode environment, and some basic notions about the EasyLocal++ tool. Although the number of users of Gecode and EasyLocal++ is growing, there exists a large community of researchers that deals with optimization problems using the *Logic Programming* paradigm. These people usually model and solve optimization problems using Constraint Logic Programming (CLP) environments that support the search on Finite Domains, such as SICStus [58] and ECLiPSe [77].

It should be desirable to use all the characteristic of the GELATO solver also starting from CLP models, instead of using just C++ models written for the Gecode environment. To this aim we developed a compiler able to translate a model encoded with a high-level CLP language $\mathcal{L}_{CLP}$ into a low-level encoding $\mathcal{L}_{ll}$. $\mathcal{L}_{ll}$ is a language that Gecode can easily read and interact with. As a consequence, a model encoded with the $\mathcal{L}_{ll}$ language can be easily read and solved using GELATO, via the Gecode environment. In this way, a programmer can exploit the high level of abstraction of the CLP modeling language to model a problem, and then use an effective low-level solver to explore the search space with LNS. In our current implementation we use SICStus Prolog as $\mathcal{L}_{CLP}$ high level language and FlatZinc as $\mathcal{L}_{ll}$ target language. Moreover, any language provided with a compiler to the FlatZinc language can be used as $\mathcal{L}_{CLP}$ language. Examples of these languages are MiniZinc, with the compiler presented in [57] as well as Haskell, with the back-end provided in [84].

SICStus Prolog and FlatZinc are presented respectively in Section 8.1 and Section 8.2, with focus on the characteristics that make them particularly suitable to our aims. Then, a description of the compiler and of the compiling process is given in Section 8.3 . In the last sections a meta-modeling language for LNS is introduced (8.4) and an overall vision of the GELATO tool is given (8.5).

8.1 SICStus language

SICStus Prolog [58] is a state-of-the-art implementation of the Prolog language. It follows the ISO standard and provides a wide range of functionalities, e.g., debugger, interactions

with the OS, bi-directional interfaces to programming languages (such as C, C++, .NET, Java), as well as a huge number of built-in predicates and data structures. Moreover, SICStus Prolog implements a library for Constraint Logic Programming over Finite Domains (i.e., `library(clpfd)`), that allows to encode CSPs and COPs in a declarative way and explores the search space with different kinds of engines. This library also implements a wide variety of global constraints. Thanks to these characteristics, SICStus Prolog is nowadays one of the most used CLP system by the community of logic programmers. Other systems that provide similar functionalities and are also well known in the LP community are ECLiPSe [77], SWI-Prolog [80, 81], and B-Prolog [85].

We choose SICStus as our high level modeling language because of its popularity among the users. It is also worth noting that SICStus and the other Prolog implementations (such as ECLiPSe, B-Prolog, etc.) share the same syntax, with just slight differences. Thus, all the work that is going to be presented in this chapter can be easily replicated for other Prolog implementations.

In the next sections an explanation about how to model and solve CSPs and COPs with SICStus Prolog is given, suggesting to look at [43, 48] for an exhaustive introduction to logic programming and constraint logic programming, as well as at [58] for a deep explanation of the SICStus Prolog language.

8.1.1 CSPs and COPs in SICStus

The `library(clpfd)` of SICStus prolog fully supports the modeling of CSPs and COPs as defined in section 1.1 and provides the search engines of the kind explained in 2.3. In order to explain the modeling and solving steps for SICStus Prolog, we consider the SCTT problem and the instance introduced in section 1.2. The code in figures 8.1 and 8.2 show a SICStus model for SCTT.

Main program Lines 1-2 load respectively the libraries `lists` and `clpfd`. The first library provides built-in predicates to manipulate lists and access their elements. The second one supports the definition of variables, domains and constraints, as well as the use of search engines. Then, lines 4-21 define the main predicate, `sctt`. Arguments `OS`, `PL` and `AI` are output arguments. Arguments `UnOS`, `UnPL`, `UnAI`, `L` and `D` are input arguments that represent the unavailable shifts, the required number of lectures (function l) and the minimum number of working days (function δ) for each course. Lines 5-7 define the variables (three lists of length 10), and lines 9-10 collect them together in the list `Vars`. The predicate `domain` at line 11 sets the domains of all the variables to $\{0, 1\}$. Lines 13-17 post the constraints on the variables and line 19 builds the objective function, storing it into the variable `FOBJ`. Predicate `labeling` at line 21 sets up the search engine. Its first argument consist in a list of search options: in particular, the `minimize(FOBJ)` option activates a branch and bound engine that will minimize the value of the variable `FOBJ`. The second argument specifies the variables that the search engine has to explore, in our case collected in the list `Vars`.

```
(1)     :-ensure_loaded(library(lists)).
(2)     :-ensure_loaded(library(clpfd)).
(3)
(4)     sctt(OS,PL,AI,UnOS,UnPL,UnAI,L,D):-
(5)        length(OS, 10),
(6)        length(PL, 10),
(7)        length(AI, 10),
(8)
(9)        append(OS,PL,OSPL),
(10)       append(OSPL,AI,Vars),
(11)       domain(Vars, 0, 1),
(12)
(13)       constraint1([OS,PL,AI],L ),
(14)       constraint2( OS,PL ),
(15)       constraint3( OS,UnOS ),
(16)       constraint3( PL,UnPL ),
(17)       constraint3( AI,UnAI ),
(18)
(19)       build_fobj([OS,PL,AI],D,FOBJ),
(20)
(21)       labeling([minimize(FOBJ)],Vars).
(22)    constraint1([Row|Tr],[L|Tl]):-
(23)       sum( Row, #=, L ),
(24)       constraint1(Tr,Tl).
(25)    constraint1([],[]).
(26)
(27)    constraint2([X1|T1],[X2|T2]):-
(28)       X1 * X2 #= 0,
(29)       constraint2( T1, T2 ).
(30)    constraint2( [], [] ).
(31)
(32)    constraint3( Row, [Un|Tu] ):-
(33)       nth1(Un,Row,X),
(34)       X#=0,
(35)       constraint3(Row,Tu).
(36)    constraint3(_,[]).
```

Figure 8.1: SICStus Prolog model for the SCTT problem

Constraints definitions In lines 22-36 the predicates that post the constraints (C1), (C2) and (C3) of section 1.2 are defined. The formula $\sum_{j \in P} x_{c,j} = l(c)$ for (C1) is modeled with the built in predicate `sum( Row, #=, L )`, where `Row` is a list containing the variables $x_{c,j}$ and `L` is $l(c)$. Similarly, the constraint (C2) $x_1 \cdot x_2 = 0$ is concretely rendered by `X1 * X2 #= 0` and the constraint (C3) $x = 0$ by `X #= 0`.

Objective function definition Lines 37-56 model the objective function defined in section 1.2, i.e., $fObj = \sum_{i=1}^{n} \max(0, \delta(c_i) - |\{d(j) : x_{c_i,j} > 0\}|)$. The objective function is stored into the variable `FOBJ`. The single contribution to the objective function of each course (i.e., given i, $\max(0, \delta(c_i) - |\{d(j) : x_{c_i,j} > 0\}|)$) is calculated by the predicate `single_contribution` (lines 43-46), and then recursively added to the other ones. Predicate `single_contribution` delegates to `number_of_days` the calculus of the working days of a course (i.e., $|\{d(j) : x_{c_i,j} > 0\}|$).

8.2 FlatZinc **language**

FlatZinc is a low level language developed by the NICTA research group for the G12 constraint programming platform [78]. The G12 project defines three languages for specifying combinatorial and optimization problems:

```
(37)   build_fobj([Row|T],[D|Td],FOBJ):-
(38)      build_fobj( T, Td, FObjT ),
(39)      single_contribute(Row,D,FObjR),
(40)      FObjT + FObjR #= FOBJ.
(41)   build_fobj([],_,0).
(42)
(43)   single_contribute(Row,D,FObjR):-
(44)      number_of_days( Row, Days),
(45)      Days+Diff #= D,
(46)      max( Diff, 0 ) #= FObjR.
(47)   number_of_days([M,P|T],TotalDays):-
(48)      number_of_days( T, Days ),
(49)      this_day( M, P, TD ),
(50)      TD + Days #= TotalDays.
(51)   number_of_days( [], 0 ).
(52)
(53)   this_day( M, P, TD ):-
(54)      M+P #= Sum,
(55)      min( 1, Sum ) #= TD.
```

Figure 8.2: Objective function of the SCTT model

1. *Zinc*, a high-level, typed, mostly first-order, functional modeling language; Zinc extends OPL and moves closer to CLP languages such as SICStus and ECLiPSe;
2. *Minizinc*, a medium-level, typed, purely first-order, functional modelling language; it is a subset of Zinc and provides reasonable modelling capabilities while being simpler to implement.
3. FlatZinc, a low-level input language; it is a subset of MiniZinc and is intended to be a target language for Zinc and MiniZinc. At this level, we don't have anymore the high level characteristics of MiniZinc, such as data/model separation, multi-file models, variable declaration/assignment separation, for loops, if-then-else expressions. FlatZinc is designed to specify problems at the level of an interface to a CP solver that supports finite domain and linear programming constraints.

The NICTA group provides a compilator that can translate a Minizinc model into a FlatZinc encoding, as well as a solver able to read and solve a FlatZinc file [57]. Recently, several CP environments, such as SICStus, ECLiPSe and Gecode, have been interfaced with FlatZinc.

8.2.1 CSPs and COPs in FlatZinc

A FlatZinc model for a CSPs or a COPs is a list of single statements that define variables, domains and constraints, as well as the search strategy. It is made of 5 parts: 1) zero or more external predicate declarations (i.e., a non-standard predicate that is supported directly by the target solver); 2) zero or more parameter declarations (i.e., constants values); 3) zero or more variable declarations; 4) zero or more constraints; 5) a solve goal. These five parts have to be declared in this order. Here we focus on parts 3)-5) giving examples of the FlatZinc statements.

Variables and domains To specify an integer variable X with a domain between Min and Max the following statement is used:

```
var Min..Max: x;
```

Thus, in a model for the SCTT problem, all the 30 variables are declared with the statement `var 0..1: xn;`.

Constraints A constraint statement in FlatZinc is marked with the keyword `constraint`, followed by a predicate that specify the constraint and the variables involved. For example, equalities and inequalities can be expressed as follows:

```
constraint int_eq(y, 0);      % y <= 0
constraint int_ge(0, x);      % 0 >= x
constraint int_lt(x, y);      % x < y
```

Also linear equalities and inequalities are supported. For example, the formula $2x_1 - 3x_2 + x_3 + 5x_4 = 10$ can be encoded as:

```
constraint int_lin_eq( [2,-3,1,5], [x1,x2,x3,x4], 10);
```

In [56] a comprehensive discussion of all the constraints supported by FlatZinc is given.

Search engines A model should finish with a solve statement, taking one of the following forms:

- `solve satisfy;` that search for any satisfying assignment;
- `solve minimize fObj;` that search for an assignment that minimizing the variable `fObj`;
- `solve maximize fObj;` that search for an assignment that maximize the variable `fObj`.

A solution consists of a complete assignment, where all variables in the model have been given a fixed value.

8.2.2 Annotation in FlatZinc

Annotations are optional suggestions to the solver concerning how individual variables and constraints should be handled (e.g., a particular solver may have multiple representations for int variables) and how search should proceed. An implementation is free to ignore any annotations it does not recognize. Annotations are unordered and idempotent: annotations can be reordered and duplicates can be removed without changing the meaning of the annotations.

Annotations starts with two colons `::`, followed by a string representing the annotation name. An annotation can contain arguments, enclosed into brackets. Moreover, an argument can be an annotation, allowing to nest annotations inside other ones. Thus, an annotation is:

```
:: annotation_name | :: annotation_name( arg1, arg2, ... argn )
```

For example, annotations can help to distinguish between the main variables of the model and temporary variables, used just to perform operations. A solver may want to print on the console only the value of the main variables, avoiding to return the values of the temporary ones. To this aim, FlatZinc defines the annotations `output_var` for the main variables and `var_is_introduced` for temporary ones. This annotations must be used in the statements declaring the variables, as follows:

```
var 0..1: x     :: output_var ;
var 0..1: xtemp :: var_is_introduced;
```

FlatZinc also defines annotations that specify the search strategy to perform. This annotation will be contained in the `solve` statement and is defined as follows:

```
:: int_search( vars, varchoiceann, assignmentann, strategyann )
```

where `vars` is the array of the variables to be assigned, `varchoiceann` is a nested annotation specifying the variable selection strategy, `assignmentann` is a nested annotation specifying the value selection strategy and the `strategyann` is a nested annotation specifying the kind of search. For example, a branch and bound search engine that explores variables $x1, x2$ and $x3$, minimizing variable f, with a first fail selection rule and a random value selection, should be declared as follows:

```
solve :: int_search( [x1,x2,x3,x4], first_fail, indomain_random,
complete) minimize f;
```

In section 8.4 we use FlatZinc annotations to define a LNS meta-heuristic language supported by the GELATO solver.

8.3 Translation from Prolog to FlatZinc

Starting from the compiler presented in [16], we developed a translator from a Prolog model to a FlatZinc model. The translation process works in two phases. In the first phase, that we call *compilation*, the original constraint logic program that encodes the model is parsed and transformed into a new logic program, where all the predicates involving constraint programming tasks are replaced by special *output predicates*. The new logical program is called *generator*, since it will actually generates the final FlatZinc model. In the second phase, called *generation*, the generator program will create the final FlatZinc model and the output predicates will write all the FlatZinc statements about variables, domain, constraints and search. These two steps are performed by two different logic programs written in SICStus Prolog. Note that, unlike the compiler in [16], neither intermediate ad-hoc languages, nor programming languages other than SICStus Prolog have been used. Figure 8.3 shows this two steps process. In the next sections the two phases are analyzed in more detail.

8.3.1 Compiling a FlatZinc Generator

In this first phase, a compiler translates the Prolog model into the corresponding generator. Before analyzing the compilation process, it is worthy to describe the structure of a generator. A generator is made up of three parts:

Figure 8.3: The overall GELATO tool and the translation process.

```
(1)     %%%Head
(2)
(3)     sics2fzn( Goal, OutputFzn ):-
(4)        (
(5)        open(OutputFzn, write, FZStream),
(6)        assert( flatZincStream(FZStream) ),
(7)        call( Goal ),
(8)        retract( flatZincStream(_) )
(9)           -> close(FZStream)
(10)          ; close(FZStream),
(11)          write('A problem occurred!')
(12)       ).
(13)    ...
```

Figure 8.4: Example of generator head

head: this part is the same for all the generators and defines the main predicate. The user will call this main predicate in order to generate the final FlatZinc model.

body: this part is strictly related to the specific model we are translating. It mirrors the original Prolog program, but it does not contain any predicate of the `clpf(fd)` library. All of them have been replaced by output predicates.

utilities: this part contains some auxiliary predicates that the generator needs during the generation phase. This part, as the head, is the same for all the generators.

Figure 8.4 shows the *head* of a generator, where the main predicate, `sics2fzn(Goal,OutputFzn)` is defined. It takes as input a goal of the original Prolog model and the name of the FlatZinc file to create. Line 5 opens the specified file (creating it if needed) in a `write` mode and sets up the output stream `FZStream`. This stream will receive all the output information and write them on the specified file. Then, `FZStream` is asserted in the program (line 6) via the fact `flatZincStream(FZStream)`. In this way, `FZStream` acts like a global variable, that can be easily retrieved and used at any point of the program. Line 7 executes the goal specified in the `Goal` argument. The execution of the goal is the core of the overall translating process, since it starts the generation phases (see Section 8.3.2 below). Line 8 retracts the global variable `FZStream` asserted in line 6. Lines 5-8 are inserted into a control structure that tolerates possible errors. If line 7 is executed without any problem, then line 9 is executed (`->` branch), so the output stream is closed and the execution ends. If some problem occurs, the `;` branch is executed: the output stream is closed, a message is printed and the execution ends.

```
(102)  ...
(103)  %%%Utilities
(104)
(105)  putOnes([1|T]):-
(106)     putOnes(T).
(107)  putOnes([]).
(108)
(109)  processDomain([H|T],Min,Max):-
(110)     flatZincStream(FZStream),
(111)     format(FZStream, "var ~w .. ~w: ~w :: output\_var; \n",[Min,Max,H]),
(112)     processDomain(T,Min,Max).
(113)  processDomain([],_,_).
```

Figure 8.5: Example of generator utilities

The *body* of the generator mirrors the structure of the original programs. All rules of the program predicates are rewritten, with the same head and almost the same body. Thus, recursive structures and calls to other predicates remain the same in the generator obtained. The only changes arises in correspondence to predicates belonging to the `clp(fd)` library, that are replaced by output predicates. An output predicate has the same meaning of the original one, but instead of actually posting the information in the constraint store, it prints the corresponding statement to the final FlatZinc model, with the proper FlatZinc syntax. Let us consider an example of translation from `clp(fd)` predicate to the output predicate. The predicate `H in Min..Max` from the `clp(fd)` library sets the variable `H` to have domain $\{Min \ldots Max\}$. It is translated into the following output predicate:

```
format( FZStream, 'var ~w .. ~w: ~w ::output_var;\n', [Min,Max,H] )
```

The predicate `format` is a built-in predicate in SICStus Prolog. It writes in the specified stream (in this case `FZStream`) the text passed as secund argument, substituting the characters `~w` with the values of the variables specified in the third argument. Thus, in the generation phases, the above predicate will print on the FlatZinc file the following line:

```
var Min .. Max: H ::output_var;
```

where `Min`, `Max` and `H` are substituted with the actual values. Similarly, the other `clp(fd)` predicates have been translated into output predicates, as shown in Table 8.1.

Figure 8.5 shows two *utilities*. The first one, `putOnes`, unifies the variables of a list with `1` constants. The second utility, `processDomain` takes as input a list of variables and two integers, and generate the output predicates that set the domains for each variable of the list (line 111).

Compiling the head and the utilities is straightforward, since these parts do not change from a generator to another one. Thus, we focus our analysis on compiling the body of the generator. In figure 8.6 a pseudo code of the compilation process is given.

The original SICStus program is parsed and all the rules are collected (line 1). We remark that also facts of the program are collected, since they are rules with empty body. Then each

```
(1)     Rules <- parse( Program )
(2)     forall( Rule in Rules )
(3)     {
(4)        Head, Body <- parse( Rule )
(5)        write(Head)
(6)        forall( Predicate in Body )
(7)        {
(8)           if( Predicate is clp(fd) )
(9)           {
(10)             OutputPredicate <- constructOutputPredicate( Predicate )
(11)             write( OutputPredicate )
(12)          }
(13)          else
(14)             write( Predicate )
(15)       }
(16)    }
```

Figure 8.6: Pseudo code for the compilation of the generator body

rule is parsed and divided in the head and body part (line 4). The head is written without any modification (statement `write` at line 5), since it never contain `clp(fd)` predicates. Then each predicate of the body is analyzed: if a predicate belongs to the `clp(fd)` library, the corresponding output predicate is created and then written (lines 10-11). If it is a regular predicate, it is written without any modification (line 14). All the write actions are intended to be on the FlatZinc file, via the stream `FZStream`. The construction of the output predicates (line 10) is different for each `clp(fd)` predicate and copes with particular cases and minor technical problems.

8.3.2 Generating the FlatZinc code

In the second phase of the overall translating process, the generator is executed, simulating the execution of the original program. Thus, the predicate `sics2fzn` (see Figure 8.4) is called. A goal of the original program is passed as argument, together with the name of the output file. For example, to generate the FlatZinc for the SCTT proglem encoded in Figure 8.1, the following command is executed:

```
sics2fzn( sctt(OS,PL,AI,[3,4],[3,4,],[7,8,9,10],[4,3,4],[]),'sctt.fzn').
```

It is worth noting that at this point we can still pass instance data as input. Once the final FlatZinc file is generated, all the instance data are inserted into that file as constants and the only way to change them is to directly modify the file.

The executions of the generator does not have any difference in terms of control flow, w.r.t. the execution of the original Prolog model. In fact the predicates in the generator body are called in the same way as the ones in the Prolog model. The difference is that during the execution of the original Prolog model, the `clpfd` predicates add information to the SICStus Prolog constraint store, while during the execution of the generator, the output

clpfd predicate model.pl (SICStus)	output predicate generator.pl (SICStus)	FlatZinc statement model.fzn (FlatZinc)
`H in Min..Max`	`format(FZStream,'var ~w .. ~w: ~w :: output_var;\n', [Min,Max,H] )`	`var Min..Max: H ::output_var;`
`all_different(List)`	`format(FZStream,'all_different : ~w;\n',[List])`	`all_different : List;`
`A#=B`	`format(FZStream,'constraint int_eq(~w, ~w);\n',[A,B])`	`constraint int_eq(A, B)`
`A+B#=C`	`format(FZStream,'constraint int_plus(~w, ~w, ~w);\n',[A,B,C])`	`constraint int_plus(A,B,C);`
`C#=min(A,B)`	`format(FZStream,'constraint int_min(~w, ~w, ~w);\n',[A,B,C])`	`constraint int_min(A,B,C);`
`labeling([ff, minimize(F)], Vars)`	`format(FZStream,'solve::int_search(~w,~w, ~w,~w)~w ~w;\n',[Vars,first_fail, indomain_min,complete,minimize,F])`	`solve::int_search(Vars, first_fail,indomain_min, complete) minimize F;`

Table 8.1: Examples of translation from clp(FD) predicates to output predicates and then to FlatZinc statements

predicates write the same information in the FlatZinc file. Some example of the statements written in the final FlatZinc file by the generator are given in Table 8.1.

In figure 8.7 we show some excerpts of the FlatZinc model obtained compiling the SCTT model in figure 8.1. Lines 1-2 declare the variables `x1225` and `x1227`, with domain $\{0, 1\}$. All the 30 variables of the model are declared like this. Then other temporary variables are declared (lines 4-6): this variables are used to bind values while performing operations. Lines 7-18 contains the constraints among the variables: linear constraints (e.g., line 7), operational constraints (e.g., lines 9,13,15,17) equalities constraint (e.g., line 11), and so on. The last line of the example (line 19), show the FlatZinc statements that set up the search strategy. A complete exploration is performed on the 30 model variables, with leftmost variable selection (`input_order`) and value selection from the smallest to the biggest one (`indomain_min`). Moreover, the exploration minimizes the objective function stored in variable `x10501`.

8.4 A Meta-heuristic modeling language

Using the FlatZinc annotations introduced in Section 8.2, we defined a language that models the possible hybrid LNS heuristics provided by the GELATO tool. The main annotation to this aim is the following:

```
:: gelato(lsStrategy, vars, percentage, firstSolStrategy,
LNSStrategy, stopCrit)
```

The arguments are defined as follows:

1. `lsStrategy`: annotation that specifies which LS algorithm to use;
2. `vars`: array of variables that will be explored by the algorithm;

```
(1)    var 0 ..  1:  x1225 ::  output_var ;
(2)    var 0 ..  1:  x1227 ::  output_var ;
(3)    ...
(4)    var int:  x10501 ::  var_is_introduced;
(5)    var int:  x10676 ::  var_is_introduced;
(6)    ...
(7)    constraint int_lin_eq([1,1,1,1,1,1,1,1,1,1],
       [x1225,x1227,x1229,x1231,x1233,x1235,x1237,x1239,x1241,x1243], 3);
(8)    ...
(9)    constraint int_times(x1225, x1245, 0);
(10)   ...
(11)   constraint int_eq(x1229, 0);
(12)   ...
(13)   constraint int_plus(x1281, x1283, x13887);
(14)   ...
(15)   constraint int_min(1, x13887, x13718);
(16)   ...
(17)   constraint int_max(x12028, 0, x11521);
(18)   ...
(19)   solve ::int_search([x1225,x1227,x1229,x1231,x1233,x1235,x1237,x1239,
                x1241,x1243,x1245,x1247,x1249,x1251,x1253,x1255,x1257,
                x1259,x1261,x1263,x1265,x1267,x1269,x1271,x1273,x1275,
                x1277,x1279,x1281,x1283],input_order,indomain_min,complete)
             minimize x10501;
```

Figure 8.7: Excerpt of SCTT FlatZinc model obtained

3. `percentage`: integer number from 1 to 100 that specify the percentage of free variables among `vars`, thus defining the cardinality of the set $|FV|$;

4. `firstSolStrategy`: annotation that specifies the variable and value selection strategy that will be used when searching for the first solution;

5. `LNSStrategy`: annotation that specifies the variable and value selection strategy that will be used when performing large neighborhood search on the free variables;

6. `stopCrit`: annotation that specifies the stop criterion for the exploration of a single large neighborhood.

The `lsStrategy` annotation is one of the following:

- `hill_climbing( maxIterations )`
- `tabu_search( tabuMin, tabuMax, maxIterations )`
- `simulated_annealing( startingTemp, coolingRate, stopTemp )`

where `maxIteration` is an integer that determines the maximum number of idle iterations; `tabuMin` and `tabuMax` are the parameters that determine the length of the tabu list; `startingTemp` is a float number representing the value of the temperature at the beginning of the search, `coolingrate` is a float number between 0 and 1, that determine the decrease of temperature at each step, and `stopTemp` is the value of temperature that will force the Simulated Annealing algorithm to stop.

The `firstSolStrategy` and the `LNSStrategy` annotations have respectively the following form:

- `firstSolStrategy( var_selection, val_selection )`
- `LNSStrategy( var_selection, val_selection ) | same_search`

where `var_selection` is one of the variable selection strategies supported by Gecode (such as `INT_VAR_NONE` or `INT_VAR_RND`) and `val_selection` is one of the variable selection strategies supported by Gecode(such as `INT_VAL_MIN` or `INT_VAL_MAX`). An exhaustive list of the Gecode strategies is reported in [69]. We also allow to use `same_search` as `LNSStrategy`, with the meaning that the value and variable selection strategies for the LNS exploration are the same used searching for the first solution.

The `stopCrit` annotation is one of the following:

- `stop_time( ms )`
- `stop_fail( n )`
- `stop_time_fail( ms, n)`

where ms is an integer value that determines the maximum number of millisecond allowed for each neighborhood exploration and n is an integer value that determines the maximum number of failures allowed for each neighborhood exploration. The third stop criterion is the combination of the first two ones.

As explained in section 7.2, the Gecode environment (and thus GELATO) is able to read and solve a FlatZinc model. Moreover, Gecode provides data structures for FlatZinc annotation and methods to parse the file and navigate through the annotations data. Thus, GELATO can read both model and the search strategy contained in the FlatZinc file, and set up the solving hybrid algorithm in the proper way.

8.5 Architecture of the GELATOMeta-Tool

In detail we have all the ingredients that allows us to build a meta-tool for combinatorial problems. We have:

- a *high level modeling language*, i.e., SICStus Prolog, that allows the programmer to easily develop declarative models for CSPs and COPs;
- a *low-level input language*, i.e., FlatZinc, that can be easily read by constraint programming environments;
- a *compiler* that can translate the information encoded in the high-level model into the low-level language;
- a *meta-heuristic language* that allows the user to specify the LNS algorithms at high level, with annotations that can be inserted in the Flatzinc model or included in the original SICStus model;
- *efficient state-of-the-art* solvers, i.e., Gecode and EasyLocal++, that can perform exploration on the model in an effective way;
- a *a hybrid tool*, GELATO, that reads FlatZinc model and exploits the functionalities of the solvers to perform the exploration of the search space. The search strategy will hybridize the constraint programming and the local search paradigms, according to the strategies specified by the annotations.

Thus, we can define a multi-paradigm tool, GELATO, that will allow the programmer to easily model a CSP/COP, define or select the meta-heuristics for the search, and automatically obtain the encoding of his/her model and of the solution strategy (possibly heuristic or meta-heuristic) that can be read and executed to solve the problem. The tool comprises three main components: the *modeling*, *translating*, and the *solving* one. First, the user can *model* a problem in a high-level language (e.g., SICStus Prolog or Minizinc), and specify the meta-heuristic he want to use to solve the problem; then, the model and the meta-algorithm defined are automatically *compiled* into the target language (i.e., FlatZinc); at the end, the encoding obtained is read, the solver program is executed and the tools interact in the way specified by the user in the modeling phase, to *solve* the instance of the problem modeled. A schema of this *modeling-translating-solving* paradigm is given in figure 8.8.

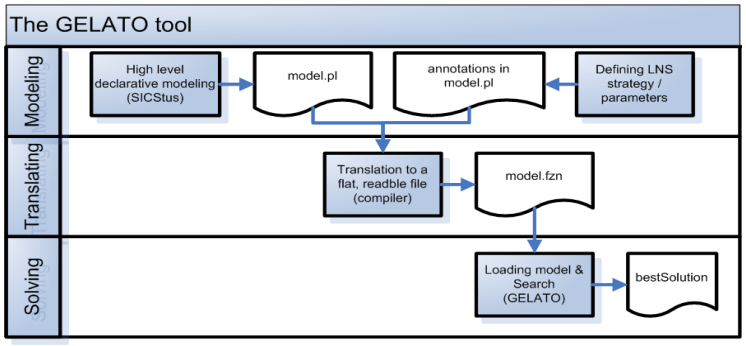


Figure 8.8: The overall GELATO tool

With such a tool, developing hybrid algorithms is flexible and straightforward and their executions will benefit from the use of low level efficient solvers, such as Gecode and EasyLocal++. As a side effect, this programming framework represents a *new way to run declarative models*: the user can encode the problems with well-known declarative languages (e.g., SICStus Prolog), and then make use of new low-level implementations (e.g., Gecode and EasyLocal++) for efficient executions. Moreover, the modeling capabilities of our tool can benefit from the front-end recently developed for the Haskell language [84] and, in general, from other front-ends to Gecode, listed in the Gecode web-site [67].

9

Installing GELATO

In this chapter instructions about how to properly install GELATO and import its functionalities into a C++ project are given. Since GELATO works on the Gecode and the EasyLocal++ tools, we first explain how to install these tools (Sections 9.1 and 9.2) and the focus on the GELATO installation (Section 9.3).

9.1 Installing Gecode

The Gecode environment is available for download from the Gecode website [67]. It is possible to download precompiled binary packages for Windows (both for the 32 and the 64 bit version) and MAC OS (32 and 64 bit), as well as source packages (to be used on Linux-like systems). This section focuses on installing the source packages, since the use of precompiled packages is straightforward. Additional information can be found in [69].

On Linux and similar operating systems Gecode is installed as a set of shared libraries. Once the source package has been downloaded and unzipped in the desired directory, it need to be configured with the `./configure` command. An example of configuration is:

```
$ ./configure CC=gcc CXX=g++ --enable-debug --enable-static
```

This line build the Gecode library statically, enabling all the assertions and information for debug, specifying to use `gcc` as C compiler and `g++` as C++ compiler. A list of all supported configuration options can be obtained by calling `./configure --help`. After the configuration, the sources can be compiled with the command `make` and then installed with the command `make install`.

Once Gecode is installed, it is necessary to set the proper option for the user compiler and linker. To compile a code using the `g++` compiler, the option `-I<dir>/include` must be added, with `<dir>` being the directory where Gecode has been installed. This option make `gcc` able to find the Gecode header files. The linker needs to know where to find the libraries so the option `-L<dir>/lib` must be given. It is also necessary to specify the libraries to link against, using the option `-l<library>`. The two libraries that always need to be linked are `gecodesupport` and `gecodekernel`. For the other libraries, the rule is that whenever a header file `<gecode/name.hh>` is included, then corresponding library must be linked with the option `-lname`.

9.2 Installing EasyLocal++

The EasyLocal++ framework can be download using a subversion command line client, with the following command:

```
$ svn co http://satt.diegm.uniud.it/svn/easylocal/EasyLocal++/tags/2.0-pre EasyLocal++
```

The code is written in standard C++ and it has been tested with the GNU C++ compiler 4 and with Microsoft Visual C++ 2005, even if the use of EasyLocal++ with the Microsoft Compiler is discouraged. Additional information can be found in [28]. As for the Gecode tool, EasyLocal++needs to be configured, compiled and installed. These three tasks can be performed with the standard commands `configure`, `make`, and `make install`.

The option for the compiler in order to properly include the EasyLocal++ header files is `-I<dir>/EasyLocalpp/src`. For `<dir>` we indicate the path on the file system to reach the directory `EasyLocalpp`, where EasyLocal++ is typically installed. The option for the linker is `-L<dir>/EasyLocalpp/lib` to specify the directory of the library and `-lEasyLocalpp`, that specify the name of the EasyLocal++ library.

9.3 Installing GELATO

Once Gecode and EasyLocal++ are properly installed, you are ready to begin the GELATO installation. First download the tool from [15]. The code is written in standard C++ and has been tested with the GNU C++ compiler 4.

The functionalities provided by GELATO are typically encoded into parametric class. This guarantees the possibility to interface GELATO to EasyLocal++ (that make a deep use of parametric classes as well) and facilitates developing new functionalities and reusing portions of code. Since in C++ parametric classes have to be stored in header files, almost the whole GELATO framework is in header files. Moreover, the not parametric remaining portion of code is very small, so we decided to keep this into header files too. In this way, there is no need to configure, compile, and install the GELATO tool. However, it is necessary to specify to the compiler where the header files are stored. Once the header file have been downloaded in the selected directory, for example `<dir>/GELATO`, the option for the compiler is `-I<dir>/GELATO`. In order to access all the functionalities of the GELATO internal core, the file `GELATOHeaders.hpp` must be included at the beginning of a C++ code.

Once these three tools are installed, the proper commands to compile and link a C++ file that uses GELATO is something similar to:

```
$ g++ -I<dir>/GELATO -I<dir>/GECODE/include -I<dir>/EasyLocalpp/src
        -c -o"mainProgram.o" "mainProgram.cpp"
$ g++ -L<dir>/GECODE/lib -L<dir>/EasyLocalpp/lib  -o"mainProgram.exe"
      mainProgram.o -lEasyLocalpp  -lgecodeflatzinc -lgecodesearch
      -lgecodeminimodel -lgecodeserialization -lgecodescheduling
      -lgecodeint -lgecodekernel -lgecodesupport -lgecodedriver
```

Of course, when using an integrated development environment, the command lines for the compiler and the linker are inserted into a `makefile`, that contains macros to correctly build the target executable.

III

Applications

10

Asymmetric Traveling Salesman Problem

In this chapter and in the following ones we show some experiments on benchmark problems, performed to test the effectiveness of the tool GELATO. We compared GELATO with the basic tool for Local Search and Constraint Programming (i.e., EasyLocal++ and Gecode), as well as with the current state-of-the-art tool for Large Neighborhood search, i.e., Comet.

Initial analysis is performed on the Asymmetric Traveling Salesman Problem, an NP-hard problem in combinatorial optimization, taken from the TSPLib [72]. Given a set of cities and the travelling costs from each city to the other ones, the Traveling Salesman Problem aims to find a roundtrip of minimal total cost, visiting each city exactly once. In the asymmetric formulation considered here, the cost to reach city i from city j may be different than the cost to reach city j from city i: this formulation is more difficult than the symmetric one, and can model several real-life situations, such as traffic, uphill and downhill roads etc. The Asymmetric Traveling Salesman Problems (ATSP) is defined as follows.

DEFINITION 10.0.1 (ATSP) *Given a complete directed graph $G = (V, E)$ and a function c that assigns a cost to each directed edge (i, j), find a roundtrip of minimal total cost visiting each node exactly once. The edge costs might be not symmetric, i.e., in general $c(i, j) \neq c(j, i)$.*

10.1 Modeling the ATSP

Since the ATSP is well established in computer science literature, we used a standard model to encode this problem.

Let $\mathcal{X} = \{x_1, \ldots, x_n\}, n = |V|$ be the set of variables with domains $D_1 = D_2 = \ldots = D_n = \{1, \ldots, n\}$. The value of x_i represents the *successor* of the vertex i in the tour. We exploit the global constraint `circuit`$([x_1, \ldots, x_n])$ available in most CP frameworks, to constrain the solutions to represent a tour. The objective function is defined as $fObj = \sum_{i=1}^{n} c(i, x_i)$.

10.2 Experiments

We tested the ATSP on the following instances of growing size, taken from the TSPLib [72]: br17 (that we call *instance 0*, with $|V| = 17$), ftv33 (*inst. 1*, $|V| = 34$), ftv55 (*inst. 2*, $|V| = 56$), ftv70 (*inst. 3*, $|V| = 71$), kro124p (*inst. 4*, $|V| = 100$), and ftv170 (*inst. 5*, $|V| = 171$). In the next sections, we explain the different solving algorithms used as well as the processes to tune the parameters and analyze the data.

10.2.1 Solving approaches

The problem instances have been solved using:

1. a pure constraint programming approach in Gecode
2. a pure local search approach in EasyLocal++, and
3. LNS meta-heuristics encoded in $\mathbb{GELATO}$ and Comet.

Let us analyze the three approaches in more detail.

CP For the sake of showing the possibility of using $\mathbb{GELATO}$ starting from already available CP models, we used the ATSP model reported in the set of Gecode examples, slightly adapted for its integration into $\mathbb{GELATO}$. After trying different search strategies we chose those with better performances: the variable with smallest domain is selected, and the values are chosen in increasing order.

LS The pure LS approach in EasyLocal++ is based on an elementary move definition, called *1-Opt* according to the TSP terminology: given a tour, two nodes are randomly selected and swapped. We used a Randomized Hill Climbing algorithm, i.e., at each step a swap-move is randomly chosen and it is selected only if it shows improvement. The algorithm is stopped after 1000 idle iterations. We choose the Hill Climbing scheme in order to compare LS and LNS on a fair basis, since LNS is basically a "large" hill climbing approach, as explained in the next paragraph. It is worth noting that for LNS implementation we can directly reuse the existing Gecode models, whereas in using EasyLocal++ we had to implement from scratch the basic LS classes for the problem.

LNS The LNS meta-heuristic is composed by a deterministic CP search for the first solution and then a LNS exploration based on a hill climbing algorithm with a maximum number of idle iterations. Since LNS actually performs a "large" hill climbing in a wider neighborhood, we call it mountain climbing. Given a COP $O = \langle\langle \mathcal{X}, \mathcal{D}, C\rangle, FObj\rangle$, the first solution is obtained by a CP search without any pre-assigned value of the variables, without any timeout, as well as with the variable and value selection strategy used for the pure CP approach. The Large Neighborhood definition we have chosen is as follows. Given a number $N < |\mathcal{X}|$ and a solution in $\mathbf{sol}(O)$ we randomly choose a set $FV \subseteq \mathcal{X}$ of free variables, so that $|FV| = N$. Therefore, the exploration of the neighborhood consists in the possible assignments to the constrained variables FV. Each neighborhood exploration is performed by the branch and

bound search engine (see Section 2.3) with a stop criterion based on a maximum number of failures. Thus, the exploration is regulated by the value of N (affecting the neighborhood dimension) and by the number `maxF` of maximum number of failures in an improving stage (affecting the time spent exploring a single neighborhood). At each stage variables and values selection strategies are the same as those used for the first solution.

We use the same model, the same meta-heuristics, and the same parameters for both the GELATO tool and for the Comet system, in order to guarantee a fair comparison.

All computations were run on an AMD Opteron 2.2 GHz Linux Machine. We used Gecode 3.1.0, EasyLocal++ 2.0, and Comet 2.0.1.

10.2.2 Parameters tuning and data analysis

We tried several LNS approaches, that differ on the parameters N and `maxF`. N is determined as a percentage of $|\mathcal{X}|$ (10%, 20%, 30%, etc.). For relating the number `maxF` to the exponential growth of the search space w.r.t. the number N of free variables, we calculate $\texttt{maxF} = 2^{\sqrt{N * \texttt{Mult}}}$, and allow to fix the values for the parameter `Mult`. Reasonable values for `Mult` range from 0.01 to 2. Of course, these values cannot be chosen independently of the size of the problem instance and of the problem difficulty. These values are given by optional arguments of the command-line call to the execution.

Choosing a range to analyze can be done with a few preliminary runs setting `Mult` = 1 and starting with small N (e.g., 1% and then double it iteratively) in which the number k of consecutive idle iterations is kept low (e.g., 20). For small values of N the algorithm performs little improvements at each step and, frequently a number of steps without improvements. Execution stops sooner with higher values of the objective function. With increasing values of N the running time becomes slower, but one might note the computation of better optima. At a certain point one notes that the algorithm is slower and slower, but, even worse, the optimum is not improved.

Once an interval of "good" behavior is found, one might enlarge the number k of consecutive idle iterations allowed before forcing the termination (in all our LNS tests, $k = 50$) and starting tests with different values of `Mult`.

From some preliminary experiments, we discovered that LNS does not work well with small values of N, since this determines a small sized neighborhood. Even though Gecode is able to visit the neighborhood within the allotted time, it is difficult to make improvements when too few free variables are allowed to change. Conversely, when N is large several variables are allowed to change, thus the spaces to analyze are bigger. In these spaces it is easier to find to better solutions. However, at the same time it might happen that an improving solution is hidden within a huge space and it is not reached within the number of failures or millisecond allowed for the exploration. Obviously this trade-off is dependent on the particular problem being examined and has to be investigated experimentally. However, on the basis of the analysis performed in this paper, we find out that a set of reasonable values for the number of variables and the size of the search spaces is $N = 10\%, \texttt{mafF} = 1$. We set these value as the default for our tool.

Since the Gecode approach is deterministic, we perform only one run of Gecode for each instance, with a timeout of one hour. Conversely, pure LS and LNS computations are

stochastic in nature, thus we collect the results of 20 runs in order to allow basic statistical analysis. In each run, we stored the values of the objective function that correspond to improvements of the current solution, together with the running time spent. This data has been aggregated in order to analyze the average behavior of the different LNS and pure LS strategies on each sample. To this aim we performed a discretization of the data in regular time intervals; subsequently, for each discrete interval we computed the average value of the objective function. For ATSP the range of parameter we have tested is the following: $N \in \{20\%, 25\%, 30\%, 35\%, 40\%, 45\%\}$ and $\texttt{Mult} \in \{1, 1.5, 2\}$. We experimentally determine that for ATSP the best parameter combinations is $N = 35\%$ and $\texttt{Mult} = 2$.

10.3 Results

In order to compare the various solvers, i.e., Gecode, EasyLocal++, GELATO, and Comet, we shows the result of the solvers on the six ATSP instances (see Figure 10.1).

For GELATO and Comet we report the results obtained with the best parameters combination ($N = 35\%$ and $\texttt{Mult} = 2$). In these figures, the horizontal dotted line represents the best known solution for the instance considered.

It is clear from the results that pure CP (Gecode) is unsuitable to improve the objective function and to reach a satisfactory solution in reasonable time. Indeed, after some improvements in the very first stage, CP is not able to find better solutions (as the almost horizontal line shows). The CP engine is exploring systematically the whole search space and is unable to extend the searches to furthest regions. Nevertheless, Gecode is very useful to provide a good starting point for the LS and LNS approaches.

Conversely, the pure hill-climbing method (implemented in EasyLocal++) is fast, finding appropriate solutions in few seconds. The visual representation indicates rapid and substantial improvements, with a near vertical plot of the objective function, which reaches values comparable to the one reached by the LNS approaches. While in some instances the LS method is the most effective, in difficult instances (such as the last one attempted) it is outperformed by GELATO and Comet.

LNS approaches are more regular, in that they improve the objective function more slowly than LS, but with a constant trend. Comparing the two LNS approaches, it is noticeable that GELATO is faster than Comet. The graphs shows the solutions found by GELATO are always below the Comet ones, and the final solution reached is always better or equal to the one obtained by Comet. We remark that the LNS algorithm and the search strategy are the same for GELATO and Comet, in order to provide a fair comparison. In general, LNS is the most effective approach. In difficult cases it finds better solutions than hill-climbing, because it can perform a deeper search and fall later into more global local-minima.

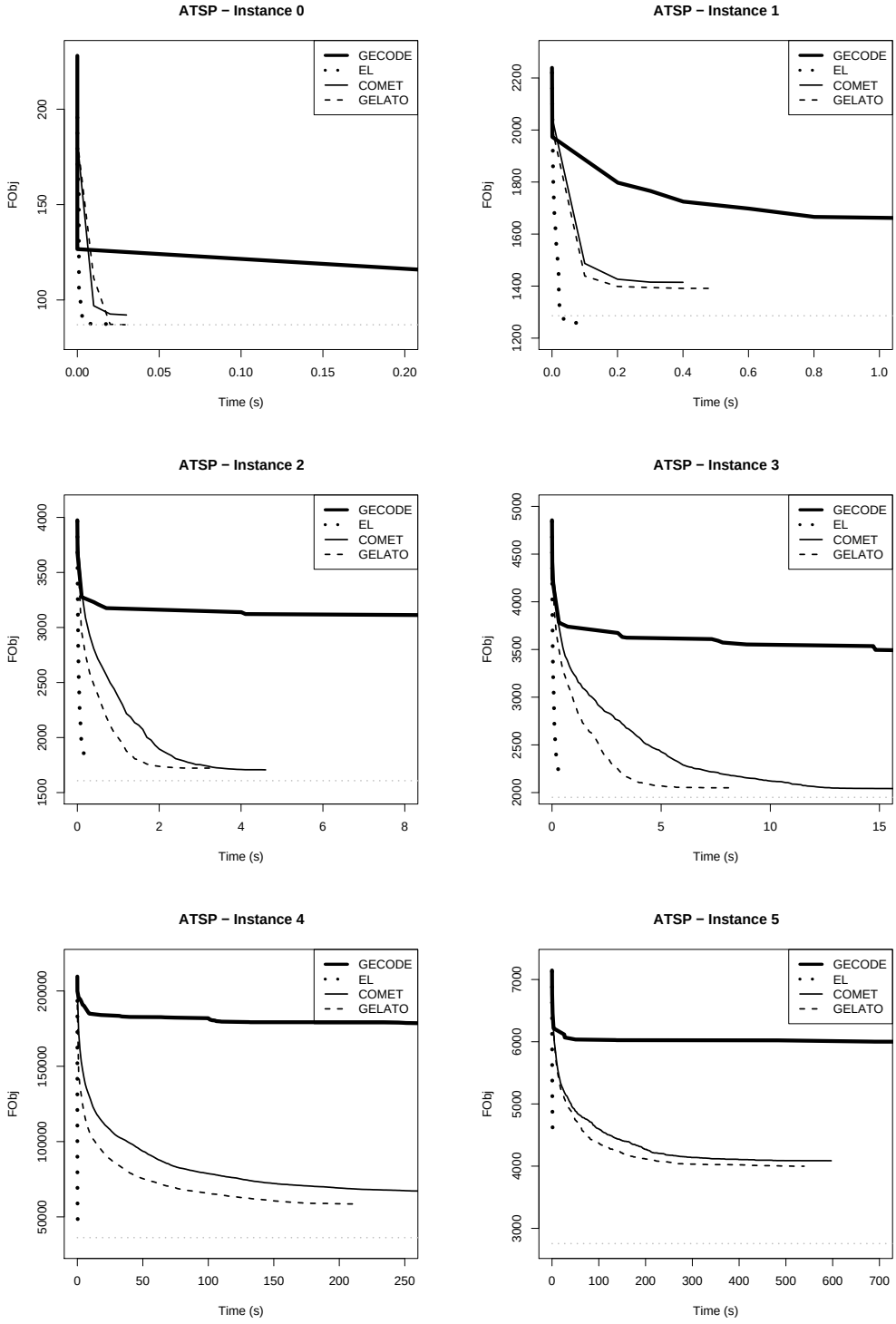


Figure 10.1: Methods comparison on the ATSP instances

11

The Minimum Energy Broadcast Problem

The Minimum Energy Broadcast Problem (MEB) is drawn from CSPLib [40] where is listed as problem number 48. It is an NP-hard optimization problem for the design of ad hoc Wireless Networks.

An ad hoc Wireless Network is a collection of wireless devices that form a network without any centralized infrastructure. When the devices are deployed they must first configure themselves to form a correctly functioning network. One configuration task when operating in a battery limited environment is the Minimum Energy Broadcast (MEB) problem. Assume a network of devices with omnidirectional antennas. The aim is to configure the power level in each device such that if a specified source device broadcasts a message it will reach every other device either directly or by being retransmitted by an intermediate device (a broadcast tree is formed). The desired configuration minimizes the total energy required by all devices, thus increasing the lifetime of the network.

Further information can be found in [82]. The formal definition for the Minimum Energy Broadcast Problem is the following.

DEFINITION 11.0.1 (MEB) *Given a set of n nodes $V = \{1, \ldots, n\}$ forming a complete graph K_n, a source node $s \in V$ and a cost function $p : V \times V \to \mathbb{R}$, representing the transmission cost between two nodes, the problem consists in finding a (directed) spanning tree rooted at s that minimizes a cost function f, which measures the energy needed by a node for broadcasting information. The function f is defined as follows: assume a tree τ is given, and let us denote by $\texttt{next}(i)$ the set of nodes reachable from a node i in τ; then $f(\tau) = \sum_{i=1}^{n} \max\{p(i, j) : j \in \texttt{next}(i)\}$.*

11.1 Modeling the MEB

We model the problem using a set $\mathcal{X} = \{x_{i,j} \in \{0, 1\} | i, j \in V\}$ of n^2 Boolean variables, with the meaning that $x_{i,j} = 1$ if and only if node i communicates with node j in the solution. Let us define the function $\delta : V \to V$ as follows: $\delta(j) = s$ if $j = s$, and $\delta(j) = i$ if $j \neq s$ and $x_{i,j} = 1$ (the function is well-defined as soon as we know that for all $j \neq s$ exists exactly one

i such that $x_{i,j} = 1$). Then, for $k \in \mathbb{N}$, $\delta^k(i)$ is recursively defined as follows: $\delta^0(j) = j$ and $\delta^{k+1}(j) = \delta(\delta^k(j))$.

The constraints added to the problem to ensure the tree structure of the output are the following:

$$\sum_{i=1}^{n} x_{i,j} = 1 \qquad j = 1, \ldots, n, j \neq s \qquad \text{(H1.1)}$$

$$\sum_{i=1}^{n} x_{i,s} = 0 \qquad \text{(H1.2)}$$

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j} = n - 1 \qquad \text{(H2)}$$

$$\delta^n(j) = s \qquad j = 1, \ldots, n \qquad \text{(H3)}$$

and the objective function is: $f = \sum_{i=1}^{n} \max_{j=1}^{n}(x_{i,j}, p(i, j))$.

As a minor implementation note, we observe that we have multiplied input data by 100 so as to use (finite) integer values for p instead of the real values stored in the problem instances used.

11.2 Experiments

For the MEB problem we selected the following six instances of size $|V| = 20$ (hence, $|\mathcal{X}| = 400$) from the set used in [82]: p20.02, p20.03, p20.08, p20.14, p20.24, p20.29.

11.2.1 Solving approaches

As for the ATSP, we solved the MEB using Gecode, EasyLocal++, $\mathbb{GELATO}$ and Comet. Now we give some more detailed information about the various approaches.

CP Differently from the ATSP case, for MEB we encoded the Gecode model from scratch. The CP strategies chosen, since they were showing the best performance, are the following: the variable with smallest domain is selected and the values are chosen randomly. Let us observe that even if some random choices are used, this randomness of Gecode is in fact deterministic.

LS For MEB we define a *change-parent-move*: given a directed rooted tree, two nodes A and B are selected, so that B is not an ancestor of A; then A becomes the parent of B. The LS algorithm used is a Randomized Hill Climbing (as for the ATSP): at each step a change-parent-move is randomly chosen and it is selected only if it is an improving move. The algorithm is stopped after 1000 idle iterations.

LNS The same LNS schema used for ATSP (i.e., the mountain climbing schema) has been used for the MEB problem. Thus, the first solution is obtained by a pure CP search. The Large Neighborhood definition we have chosen is parametric to N (neighborhood dimension) and `maxF` (stop criterion). In each neighborhood exploration variables and values selection strategies are the same used for the first solution. The GELATO and Comet implementations share the same model, the same meta-heuristics, and the same parameters values.

11.2.2 Parameters tuning and data analysis

The parameters tuning follows the same approach used for the ATSP, explained in Section 10.2.2. After some preliminary tests, we chose to perform the experiments with the following parameters: $N \in \{35\%, 40\%, 45\%, 50\%\}$ and $\texttt{Mult} \in \{0.5, 0.75, 1\}$. From the experiment, we determined that the best parameter combination is $N = 50\%$ and $\texttt{Mult} = 0.5$.

As for the ATSP, we executed just one run of Gecode for each instance (because it is deterministic), and 20 runs for EasyLocal++and GELATO. The information obtained by the 20 runs has been then discretized and aggregated, as explained in section 10.2.2.

11.3 Results

Figure 11.1 shows the comparison between the four solver on six instances of the MEB problems. As in the graphs in Figure 10.1, the horizontal dotted line represents the best known solution for the instance considered. In the experiments on the MEB instances, EasyLocal++, Gecode and GELATO starts the search process from the same initial solution provided by Gecode, while Comet starts from a different one. However, the LNS algorithm implemented in GELATO and Cometshare the same exploration strategy, and the same parameters (i.e., $N = 50\%$ and $\texttt{Mult} = 0.5$).

As in the results for ATSP problem, the pure CP programming in Gecode does not lead to good improvements of the objective function. On the other side, the meta-heuristic approaches (LS and LNS) quickly find good improvements, often very close to the best value known. The pure Local Search approach is the weakest one: even if it is very fast and on instances p20.03 and p20.29 reach solutions near to the best know one, in the other four instances it falls quickly in a local minimum and stops improving the solution. Thus the solutions obtained are worse than the ones provided by GELATO and Comet.

The behavior of the two LNS implementations is similar: they both show a regular trend, that constantly decrease the objective function value, until a value very close or equal to the best known one is reached. Moreover, GELATO has a better performance than Comet: its line is always below the Comet one and the solution reached by GELATO are usually slightly better than the one reached by Comet.

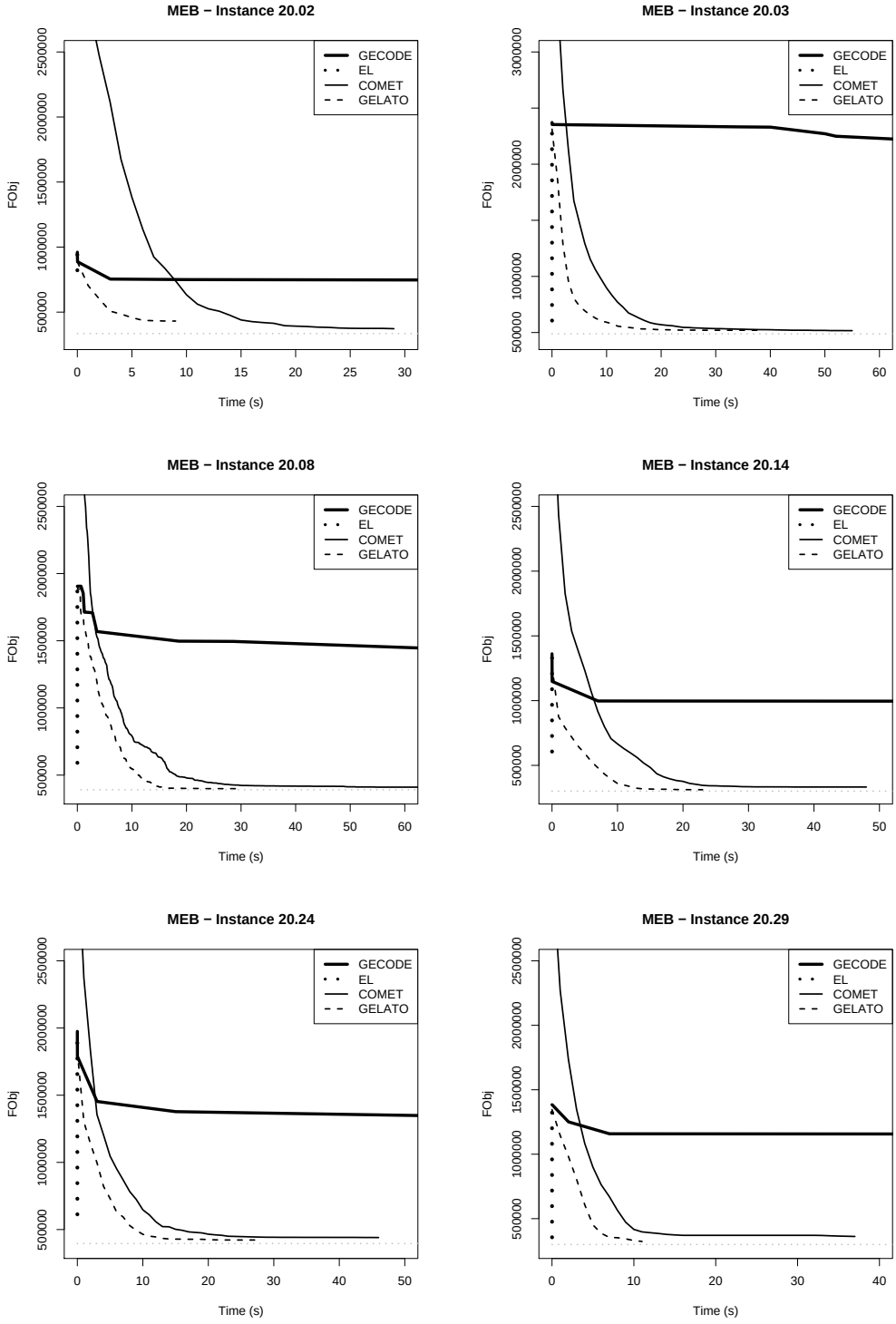


Figure 11.1: Methods comparison on the MEB instances

12

The Course Time Tabling Problem

In this chapter we present the original formulation of the Course Time Tabling Problem, that we already introduced in Chapter 1 in its simplified version (SCTT). The Course Time Tabling (CTT) problem has been introduced as Track 3 of the second International Timeta bling Competition held in 2007 [49]. It consists of the weekly scheduling of the lectures for several university courses within a given number of rooms and time periods, where conflicts between courses are set according to the curricula published by the University. This formulation applies to the School of Engineering of the University of Udine (Italy) and to many Italian and indeed International Universities, although it is slightly simplified with respect to the real problem to maintain a certain level of generality. The CTT is defined as follows.

DEFINITION 12.0.1 (CTT) *Given a set of courses $C = \{c_1, \ldots, c_n\}$, each course c_i consists of a set of lectures $L_i = \{l_{i_1}, \ldots, l_{i_a}\} \in \mathcal{L}$, is taught by a teacher $t : C \to T = \{t_1, \ldots, t_g\}$, it is attended by a number of students $s : C \to \mathbb{N}$, and belongs to one or more curricula $Q = \{q_1, \ldots, q_b\}$, where $q_i \subseteq C, i = 1, \ldots, b$, which are subset of courses that have students in common. Moreover it is given a set of periods $P = \{1, \ldots, p\}$, each period belonging to a single teaching day $d : P \to \{1, \ldots, h\}$, and a set of rooms $R = \{r_1, \ldots, r_m\}$, each with a capacity $w : R \to \mathbb{N}$. Each teacher t_i can be unavailable for some periods $u : T \to 2^P$.*

The problem consists in finding an assignment $\tau : \mathcal{L} \to R \times P$ of a room and period to each lecture of a course (H1) so that lectures of courses in the same curriculum (H3.2) or taught by the same teacher (H3.1) are scheduled in distinct periods, teacher unavailabilities are taken into account (H4), and for each (room, period) pair, only one lecture is assigned (H2).

Moreover, a cost is defined for the following criteria (soft constraints): (S1) each lecture should be scheduled in a room large enough for containing all its students; (S2) the lectures of each course should be spread into a given minimum number of days $\delta : C \to \{1, \ldots, h\}$; (S3) lectures belonging to a curriculum should be adjacent to each other; (S4) all lectures of a course would be preferably assigned to the same room.

12.1 Modeling the CTT

We model the CTT problem by defining the set $\mathcal{X}$ of $C \cdot P$ variables $x_{c,p} \in \{0, \ldots, r\}$, with the intuitive meaning that $x_{c,p} = r > 0$ if and only if course c is scheduled at period p in room r, and $x_{c,p} = 0$ if course c is not scheduled at period p. The constraints are modeled as follows:

$$\sum_{j \in P} (x_{c_i,j} = 0) = p - |L_i| \qquad i = 1, \ldots, n \tag{H1}$$

$$\sum_{i=1}^{n} (x_{c_i,j} = 0) \geq n - m \qquad j = 1, \ldots, p \tag{H2}$$

$$x_{c,j} \cdot x_{c',j} = 0 \qquad c, c' \in C \text{ s.t. } t(c) = t(c'),\ j = 1, \ldots, p \tag{H3.1}$$

$$\sum_{c \in q_i} x_{c,j} \leq 1 \qquad i = 1, \ldots, b,\ j = 1, \ldots, p \tag{H3.2}$$

$$x_{c,j} = 0 \qquad j \in u(t(c)), c \in C \tag{H4}$$

The objective function f is the sum of the following four components:

$$s_1 = \sum_{i=1}^{n} \sum_{j=1}^{p} \max(0, s(c_i) - w(x_{c_i,j})) \tag{S1}$$

$$s_2 = 5 \cdot \sum_{i=1}^{n} \max(0, \delta(c_i) - |\{d(j) \ : \ x_{c_i,j} > 0\}|) \tag{S2}$$

$$\begin{aligned} s_3 = 2 \cdot \sum_{i=1}^{b} & \Big|\{k \in P \ : \ \mathsf{start}(k) \wedge \sum_{c \in q_i} x_{c,k} \neq 0 \wedge \sum_{c \in q_i} x_{c,k+1} = 0\}\Big| + \\ & \Big|\{k \in P \ : \ \mathsf{end}(k) \wedge \sum_{c \in q_i} x_{c,k} \neq 0 \wedge \sum_{c \in q_i} x_{c,k-1} = 0\}\Big| + \\ & \Big|\{k \in P : \mathsf{mid}(k) \wedge \sum_{c \in q_i} x_{c,k} \neq 0 \wedge \sum_{c \in q_i} x_{c,k-1} = 0 \wedge \sum_{c \in q_i} x_{c,k+1} = 0\}\Big| \end{aligned} \tag{S3}$$

$$s_4 = \sum_{i=1}^{n} (|\{r \ : \ \exists p (x_{c_i,p} = r)\}| - 1) \tag{S4}$$

where start, end, and mid are three Boolean functions that state if a period is initial, ending, or in the middle of a day, respectively.

12.2 Experiments

For the CTT problem we selected from http://tabu.diegm.uniud.it/ctt/ six instances with different features (instance size, average number of conflicts, average teacher availability, average room occupation, ...): comp01, comp04, comp07, comp09, comp11, comp14.

12.2.1 Solving approaches

We solved the CTT with Gecode, EasyLocal++, GELATO and Comet.

CP As for the MEB problem, we encoded the Gecode model of CTT from scratch. After having tried several different search strategies, we obtained the best performances with the following combination: the most constrained variable is selected, breaking ties randomly, and values are chosen randomly. As previously said, the randomness of Gecode is in fact deterministic.

LS For CTT we use a *time-and-room* exchange move. Given a timetable, we randomly select a lesson scheduled at period p in room r and move it to another period p_1 into another room r_1, chosen from the empty ones. Also for the CTT we used the Randomized Hill climbing algorithm, with the limit of 1000 idle iterations.

LNS The LNS algorithm used is the same used for ATSP and MEB, i.e. the mountain climbing schema: first solution is obtained by a CP search using Gecode; Large Neighborhood definition parametric to N (neighborhood dimension) and `maxF` (stop criterion); each neighborhood exploration performed with the same variables and values selection strategies used for the first solution.

12.2.2 Parameters tuning and data analysis

The parameters tuning approach is the same already explained for ATSP and MEB. We focused the experiments on the following promising parameters: $N \in \{2\%, 3\%, 4\%, 5\%, 10\%, 15\%\}$ and `Mult` $\in \{0.1, 0.5, 1\}$. We obtained the best results from the following parameter combination: $N = 5\%$ and `Mult` $= 0.5$.

As for the ATSP and MEB, we executed just one run of Gecode for each instance, and 20 runs for EasyLocal++ and GELATO. The information obtained by the 20 runs has been then discretized and aggregated, as explained in section 10.2.2. The GELATO and Comet implementations share the same model, the same meta-heuristics, and the same parameters values.

12.3 Results

Figure 12.1 shows the results obtained from the experiment on six instances of the CTT problem. As for the previous experiments, the horizontal dotted line represents the best known solution. Gecode and GELATOstarts the search process from the same initial solution, provided by Gecode. For EasyLocal++ we exploited the hill climbing implementation developed by the Udine team for the competition in [49]: this algorithm starts from an initial solution heuristically obtained. We report the LNS searches performed with the best parameters combination (i.e., $N = 5\%$ and `Mult` $= 0.5$).

The observations made in for the previous experiments are confirmed also by these one. Moreover, the behavior of the solvers is now clearer, since the CTT is an harder problem

(with an higher number of variables and constraints). The Gecode approach is confirmed to be the worse one in order to find improving solutions. Then comes the EasyLocal++ algorithm: it is very fast, but quickly falls into local minima and stops the search without deep improving. GELATO and Comet show comparable behaviors leading to objective function values close to the best known one. However, GELATO outperforms Comet, being faster and reaching better solutions.

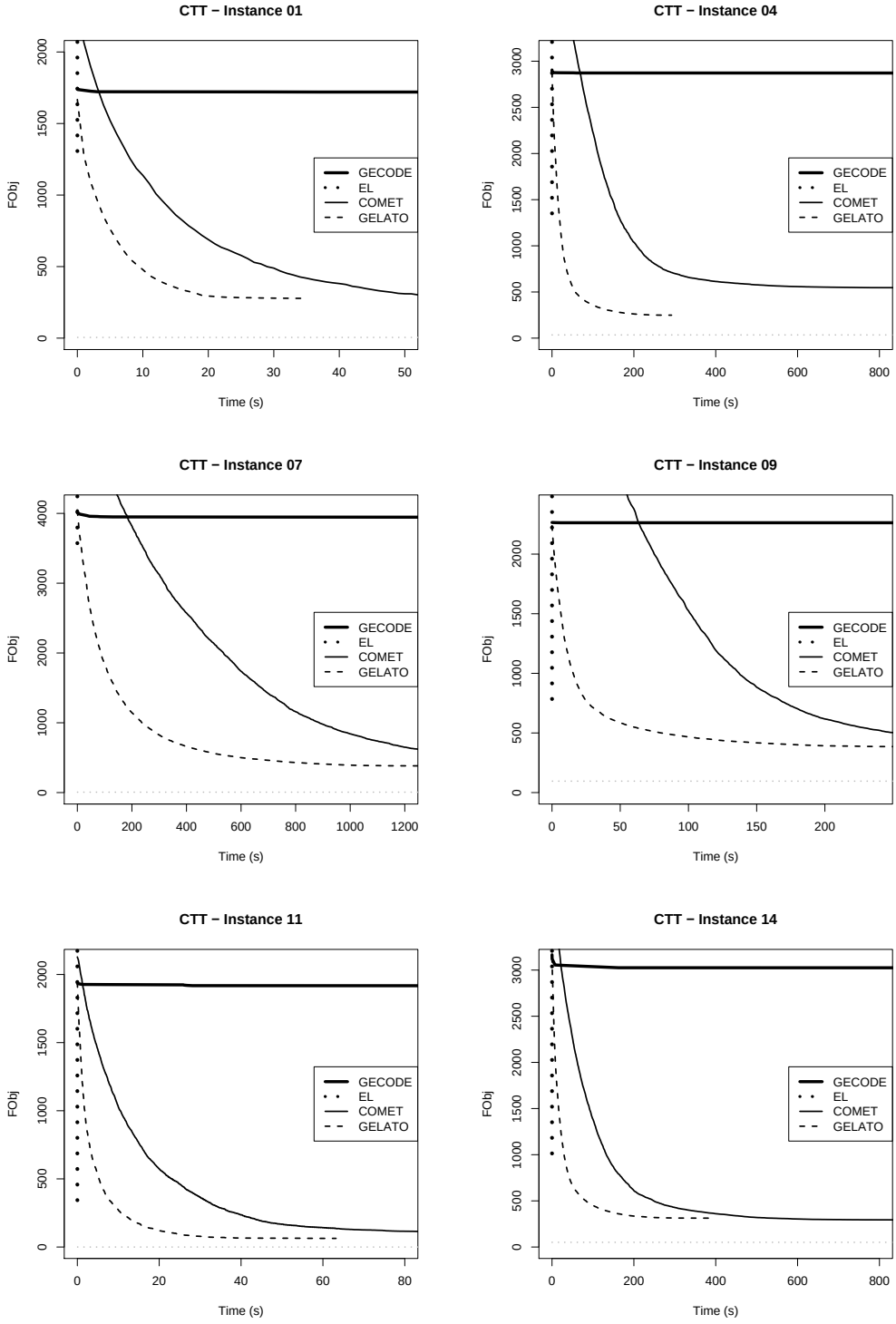


Figure 12.1: Methods comparison on the CTT instances

Conclusions

In this thesis we investigated several approaches for the solution of NP-hard problems ranging from traditional to innovative techniques. The main contribution of this thesis is the development of a multi-paradigm, hybrid tool for modeling and solving combinatorial problems.

Encouraged by the positive results of a combination of the SICStus Prolog constraint solver and Easylocal++ on a real-world application [13], we systematized our approach and developed the GELATO solver [14, 15]. GELATO is based on the state-of the art solvers Gecode and EasyLocal++ and inherits all their characteristics: it is free (downloadable from [15]), open-source, written in C++, and easy to interface to/from other systems. It provides all the functionalities of these tools, together with several new hybrid possibilities, that allow programmers to solve Constraint Satisfaction/Optimization problem in a very easy and effective way.

By using GELATO, those more accustomed to Gecode will have a library that adds local search features and allows them to easily define LS/LNS algorithms. For those who use EasyLocal++, GELATO is an extension that allows programmers to delegate to a CP engine tasks related to LS search. It is worth noting that in order to use GELATO, there is no need to learn a new language, since it is developed entirely in C++ and furthermore all the models written in Gecode or FlatZinc can be directly used in conjunction with this tool. Thus, with a single Gecode or FlatZinc model, one can exploit pure Constraint Programming, traditional Local Search and hybrid LNS algorithms with different strategies. We tested GELATO on a set of benchmark problems, and showed its effectiveness also compared to Comet, a state-of-the-art proprietary environment for (hybrid) optimization.

In order to simplify the use of GELATO for non C++ users, we made the GELATO functionalities accessible by declarative languages, such as MiniZinc, SICStus Prolog, Haskell. We developed a meta-heuristic modeling language that allows users to specify the hybrid algorithm to use together with its strategy and parameters. We also developed a compiler that can translate a SICStus Prolog model into a FlatZinc model (a preliminary version of the compiler was presented in [16]). Exploiting this and other compilers from high-level languages to the target language FlatZinc (such as the compilers presented in [57, 84]), one can model a problem in his/her favorite language and use the GELATO functionalities on the respective low-level encoding. Thus, GELATO is a multi-language tool for combinatorial optimization, which is able to deal with CSP/COP models expressed in different modeling languages and to use a combination of local search and constraint programming techniques for solving them. Being built upon Gecode, GELATO allows the user to exploit the emerging literature of problems already encoded in Gecode and use GELATO for speeding-up the search. GELATO is also built upon EasyLocal++, so the user can inherit various local search techniques from EasyLocal++, without the need of reformulating the model.

This approach also presents some limits since up to now, GELATO does not yet provide support for global constraints of the high level languages or facilities to easily model complex

multi-neighborhood Local Search approaches. Let us briefly analyze these two aspects.

Modern Prolog systems such as SICStus Prolog provide libraries for constraint programming endowed with several high-level facilities to model CSPs and COPs. For example, predicates for global constraints are usually handled with ad-hoc constraint propagators, defined in those libraries. It is not easy to translate these global constraints to the corresponding FlatZinc statements, since they are not supported by the FlatZinc language (see [56] for the complete list of constraints). Thus, when the original model contains global constraints, the use of GELATO is not as straightforward. There are several possible ways to overcome this problem: 1) remodel the problem, replacing the global constraints with simpler ones having the same meaning, and then using the normal compiler; 2) enrich the compiler, in order to recognize the global constraints and translate them into an equivalent set of simple FlatZinc statements; 3) translating the global constraints into special annotations, that will be read and correctly handled by the Gecode solver with an ad-hoc implementation (to develop in Gecode if not already present). Using approaches 1) and 2) will reduce the propagation power since a global consistency property will be replaced by a conjunction of local consistency properties. Thus, considering also that Gecode already implements several global constraints, the third approach seems the most promising.

The meta-modeling language provided by GELATO is able to define the degree of interaction between LS ans CP, thus leading to a very straightforward definition of LNS algorithms (allowing to set several combinations of strategies and parameter values). The EasyLocal++ tool also provides the possibility to define combinations of different neighborhood and/or different local search paradigms (i.e., Multi-Neighborhood Search [24]). This Multi-Neighborhood approach can be exploited in GELATO, but it requires an adept knowledge of the EasyLocal++ tool, in order to properly define the combination of the Local Search strategies. It would be desirable to extend the current meta-modeling language in order to support also the definition of such complex Local Search approaches. In this way, Multi-Neighborhood Search can easily be used by a wider community of users, including those preferring an high level declarative approach.

In the immediate we plan to do more work to enrich the compiler and the meta-modelling language. The compiler will be improved by adding the possibility to recognize and translate at least the most common global constraints. The meta-modeling language will likely be extended with structures supporting the definition of Multi-Neighborhood approaches. We also plan to develop a more clever technique for determining good candidates for the values of the tool parameters so as to allow the user to exploit GELATO as a black-box. Once all this high level features are realized, we can develop a Graphic User Interface, that allows any programmer to choose algorithms and parameters to solve a model, and access the black-box GELATO without modifying the underlying code. However, GELATO is completely written in C++, so the skilled programmer can use it as a free fully configurable search engine.

We want to apply GELATO on other challenging problems in order to gain feedback from its use and continue further improvement of the tool. In particular, we will apply GELATO on the Protein Folding problem that we dealt with in [12], where we integrated Gecode and EasyLocal++ in a direct fashion. We want to compare the performances of GELATO on the Protein Folding problem with the results recently presented in [71], where the constraint solver [18] is combined with a Simulated Annealing algorithm. Finally we intend to develop a user-oriented tutorial, downloadable from [15], that will show step by step

instructions on the use of $\mathbb{GELATO}$ at the various levels.

Bibliography

[1] Emile Aarts and Jan K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester (UK), 1997.

[2] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, Cambridge (UK), 2003.

[3] Various Authors. Cp-ai-or conference series. http://www.cpaior.org/.

[4] Roberto Battiti, Giampietro Tecchioli, and Istituto Nazionale Fisica Nucleare. The reactive tabu search. *ORSA Journal on computing*, 6(2):126–140, 1994.

[5] Russell Bent and Pascal Van Hentenryck. A two-stage hybrid local search for the vehicle routing problem with time windows. *Transportation Science*, 38:515–530, November 2004.

[6] George D. Birkhoff. Proof of the Ergodic Theorem. *Proceedings of the National Academy of Sciences of the United States of America*, 17(12):656–660, 1931.

[7] George D. Birkhoff. What is the ergodic theorem? *The American Mathematical Monthly*, 49(4):222–226, April 1942.

[8] Ivo Blöchliger and Nicolas Zufferey. A graph coloring heuristic using partial solutions and a reactive tabu scheme. *Computers & Operations Research*, 35(3):960–975, 2008.

[9] Christian Blum, Maria J. Blesa Aguilera, Andrea Roli, and Michael Sampels. *Hybrid Metaheuristics: An Emerging Approach to Optimization*. Springer Publishing Company, 2008.

[10] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35:268–308, September 2003.

[11] Wen-Chyuan Chiang and Robert A. Russell. A reactive tabu search metaheuristic for the vehicle routing problem with time windows. *INFORMS Journal on Computing*, 9(4):417–430, 1997.

[12] Raffaele Cipriano, Alessandro Dal Palù, and Agostino Dovier. Hybrid approach mixing local search and constraint programming applied to the protein structure prediction problem. In *WCB 2008 (Workshop on Constraint Based Methods for Bioinformatics)*.

[13] Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier. Hybrid approaches for rostering: a case study in the integration of constraint programming and local search. In Maria J. Blesa Aguilera, Christian. Blum, Andrea Roli, and Michaels Sampels, editors, *HM 2006*, volume 4030 of *LNCS*, pages 110–123. Springer, 2006.

[14] Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier. A Hybrid Solver for Large Neighborhood Search: Mixing Gecode and EASYLOCAL++. In Maria J. Blesa Aguilera et al., editor, *Hybrid Metaheuristics*, volume 5818 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2009.

[15] Raffaele Cipriano, Agostino Dovier, and Luca Di Gaspero. Gelato: Gecode + easylocal = a tool for optimization. http://www.dimi.uniud.it/dovier/GELATO.

[16] Raffaele Cipriano, Agostino Dovier, and Jacopo Mauro. Compiling and executing declarative modeling languages to gecode. In *ICLP '08: Proceedings of the 24th International Conference on Logic Programming*, pages 744–748, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[18] Alessandro Dal Palù, Agostino Dovier, and Enrico Pontelli. A constraint solver for discrete lattices, its parallelization, and application to protein structure prediction. *Softw. Pract. Exper.*, 37:1405–1449, November 2007.

[19] Emilie Danna and Laurent Perron. Structured vs. unstructured large neighborhood search. In *Principles and Practice of Constraint Programming*, volume 2833/2003 of *LNCS*, pages 817–821. Springer, CP 2003.

[20] G. B. Dantzig and P. Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8(1):101–111, 1960.

[21] George B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In *Activity Analysis of Production and Allocation*, Cowles Commission Monograph No. 13, pages 339–347. John Wiley & Sons Inc., New York, N. Y., 1951.

[22] George B. Dantzig and Mukund N. Thapa. *Linear programming 1: introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[23] Guy Desaulniers, Jacques Desrosiers, and Marius M. Solomon. *Column Generation*. GERAD 25th anniversary. Springer, 2005.

[24] Luca Di Gaspero. *Local Search Techniques for Scheduling Problems: Algorithms and Software Tools*. PhD thesis, Dipartimento di Matematica e Informatica – Università degli Studi di Udine, Udine, Italy, 2003.

[25] Luca Di Gaspero, Johannes Gärtner, Nysret Musliu, Andrea Schaerf, Werner Schafhauser, and Wolfgang Slany. A hybrid ls-cp solver for the shifts and breaks design problem. In Maria Blesa, Christian Blum, Günther Raidl, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics*, volume 6373 of *Lecture Notes in Computer Science*, pages 46–61. Springer Berlin / Heidelberg, 2010.

[26] Luca Di Gaspero and Andrea Schaerf. Writing local search algorithms using EASYLOCAL++. In Stefan Voß and David L. Woodruff, editors, *Optimization Software Class Libraries*, OR/CS. Kluwer Academic Publisher, Boston (MA), USA, 2002.

[27] Luca Di Gaspero and Andrea Schaerf. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software — Practice & Experience*, 33(8):733–765, July 2003.

[28] Luca Di Gaspero, Andrea Schaerf, and Andrea Roli. Easylocal++ version 2.0. http://tabu.diegm.uniud.it/EasyLocal++/, 2009.

[29] Roger Eckhardt. Stan ulam, john von neumann, and the monte carlo method. *Los Alamos Science*, Special Issue dedicated to Stanislaw Ulam:131–143, 1987.

[30] George S. Fishman. *Monte Carlo: Concepts, algorithms, and applications*. Springer Series in Operations Research. Springer-Verlag, New York, 1996.

[31] Filippo Focacci, François Laburthe, and Andrea Lodi. Local search and constraint programming. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, chapter Local Search and Constraint Programming, pages 369–403. Kluwer Academic Publishers, 2003.

[32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[33] Michel Gendreau, Alain Hertz, and Gilbert Laporte. A tabu search heuristic for the vehicle routing problem. *Manage. Sci.*, 40:1276–1290, October 1994.

[34] Paul C. Gilmore and Ralph E. Gomory. A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research*, 9(6):849–859, 1961.

[35] Fred Glover. Tabu Search–Part I. *INFORMS JOURNAL ON COMPUTING*, 1(3):190–206, 1989.

[36] Fred Glover. Tabu Search–Part II. *INFORMS JOURNAL ON COMPUTING*, 2(1):4–32, 1990.

[37] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[38] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32:1195–1220, 1984.

[39] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1996.

[40] Brahim Hnich, Ian Miguel, Ian P. Gent, and Toby Walsh. Csplib: a problem library for constraints. http://www.csplib.org/, September 2010.

[41] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[42] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Application*. Morgan Kaufmann, 1 edition, September 2004.

[43] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[44] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristic. *Artificial Intelligence*, 139(1):21–45, 2002.

[45] Scott Kirkpatrick, D. Gelatt, and Mario P. Vecchi. Optimization by Simulated Annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[46] Tony Lambert, Eric Monfroy, and Frédéric Saubion. Solving strategies using a hybridization model for local search and constraint propagation. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 398–403, New York, NY, USA, 2005. ACM.

[47] Alisa H. Land and Alison G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[48] John W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1993.

[49] Barry McCollum, Andrea Schaerf, Ben Paechter, Paul McMullan, Rhyd Lewis, Andrew J. Parkes, Luca Di Gaspero, Rong Qu, and Edmund K. Burke. Setting the research agenda in automated timetabling: The second international timetabling competition. *INFORMS Journal on Computing*, 22(1):120–130, 2010.

[50] Nicholas Metropolis. The beginning of monte carlo method. *Los Alamos Science*, Special Issue dedicated to Stanislaw Ulam:125–130, 1987.

[51] Nicholas Metropolis and Stanislaw Ulam. The Monte Carlo Method. *Journal of the American Statistical Association*, 44(247):335–341, 1949.

[52] Laurent Michel and Pascal Hentenryck. The comet programming language and system. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *Lecture Notes in Computer Science*, pages 881–881. Springer Berlin / Heidelberg, 2005. 10.1007/11564751/119.

[53] Laurent Michel, Andrew See, and Pascal Van Hentenryck. Transparent parallelization of constraint programming. *INFORMS J. on Computing*, 21:363–382, July 2009.

[54] Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artif. Intell.*, 58:161–205, December 1992.

[55] Eric Monfroy, Frédéric Saubion, and Tony Lambert. On hybridization of local search and constraint propagation. In *ICLP*, pages 299–313, 2004.

[56] Nicholas Nethercote. Specification of flatzinc. http://www.g12.cs.mu.oz.au/minizinc/flatzinc-spec.pdf.

[57] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessiere, editor, *CP 2007*, volume 4741 of *LNCS*, pages 529–543, 2007.

[58] Swedish Institute of Computer Science. Sicstus prolog. www.sics.se/sicstus.html.

[59] Manfred Padberg and Giovanni Rinaldi. Optimization of a 532-city symmetric traveling salesman problem by branch and cut. *Operations Research Letters*, 6(1):1 – 7, 1987.

[60] Steven Prestwich. The relation between complete and incomplete search. In Christian Blum, Maria Aguilera, Andrea Roli, and Michael Sampels, editors, *Hybrid Metaheuristics*, volume 114 of *Studies in Computational Intelligence*, pages 63–83. Springer Berlin / Heidelberg, 2008.

[61] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[62] Andrea Schaerf, Marco Cadoli, and Maurizio Lenzerini. Local++: A c++ framework for local search algorithms. *Software: Practice and Experience*, 30(3):233–257, 2000. John Wiley & Sons, Ltd.

[63] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992.

[64] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael J. Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1998.

[65] Marius M. Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Oper. Res.*, 35:254–265, April 1987.

[66] Richard M. Stallman, Peter T. Brown, and Sullivan John. Free software foundation. http://www.fsf.org, December 2010.

[67] Gecode Team. Gecode: Generic constraint development environment. `http://www.gecode.org`.

[68] Gecode Team. Gecode reference documentation. `http://www.gecode.org/doc-latest/reference/index.html`.

[69] Gecode Team. Modeling and programming with gecode. http://www.gecode.org/doc-latest/MPG.pdf.

[70] Edward Tsang and Chris Voudouris. Fast local search and guided local search and their application to british telecom's workforce scheduling problem. Technical report, Operations Research Letters, 1995.

[71] Abu Zafer M. Dayem Ullah and Kathleen Steinhöfel. A hybrid approach to protein folding problem integrating constraint programming with local search. *BMC Bioinformatics*, 11(S-1):39, 2010.

[72] Universität Heidelberg, Institut für Informatik. TSPLIB. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.

[73] Pascal Van Hentenryck, Russell Bent, and Yannis Vergados. Online stochastic reservation systems. In *In CPAIOR'06*. Springer, 2006.

[74] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[75] Pascal Van Hentenryck and Laurent Michel. Comet. http://www.comet-online.org, January 2008.

[76] Vlado Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, January 1985.

[77] M. Wallace, S. Novello, and J. Schimpf. Eclipse: A platform for constraint logic programming. Technical report, Imperial College.

[78] Mark Wallace. G12 - towards the separation of problem modelling and problem solving. In *Proceedings of the 6th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR '09, pages 8–10, Berlin, Heidelberg, 2009. Springer-Verlag.

[79] Peter Walters. *An Introduction to Ergodic Theory*, volume 79 of *GTM*. Springer-Verlag, New York, NY, 2000.

[80] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[81] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *CoRR*, abs/1011.5332, 2010.

[82] Steffen Wolf and Peter Merz. Evolutionary Local Search for the Minimum Energy Broadcast Problem. In Jano van Hemert and Carlos Cotta, editors, *EvoCOP 2008 – Eighth European Conference on Evolutionary Computation in Combinatorial Optimization*, volume 4972 of *LNCS*, pages 61–72, Naples, Italy, mar 2008. Springer.

[83] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[84] Pieter Wuille and Tom Schrijvers. Monadic constraint programming with gecode. In *Proc. of ModRef 2009, Eighth International Workshop on Constraint Modelling and Reformulation*, Lisbon, Portugal, September 2009.

[85] Neng-Fa Zhou. B-prolog: An overview. ALP Newsletter, June 2007. www.bprolog.com.